AFRL-RI-RS-TR-2012-252

# IMPLICATIONS OF MULT-CORE ARCHITECTURES ON THE DEVELOPMENT OF MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS) COMPLIANT SYSTEMS

UNIVERSITY OF IDAHO

*OCTOBER 2012*

FINAL TECHNICAL REPORT

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved* **OMB No. 0704-0188** |

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* OCT 2012 | **2. REPORT TYPE** FINAL TECHNICAL REPORT | **3. DATES COVERED** *(From - To)* MAR 2010 – APR 2012 |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** IMPLICATIONS OF MULT-CORE ARCHITECTURES ON THE DEVELOPMENT OF MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS) COMPLIANT SYSTEMS | **5a. CONTRACT NUMBER** FA8750-10-2-0134 |
| | **5b. GRANT NUMBER** N/A |
| | **5c. PROGRAM ELEMENT NUMBER** 33140F |
| **6. AUTHOR(S)** Jim Alves-Foss, Paul Oman, Ryan Bradetich, Xiaohui He, Jia Song | **5d. PROJECT NUMBER** MILS |
| | **5e. TASK NUMBER** UI |
| | **5f. WORK UNIT NUMBER** 01 |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** University of Idaho<br>Center Secure and Dependable Systems<br>875 Perimeter Drive, MS 1008<br>Moscow, Idaho 83844-1008 | **8. PERFORMING ORGANIZATION REPORT NUMBER** N/A |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** Air Force Research Laboratory/RITA<br>525 Brooks Road<br>Rome NY 13441-4505 | **10. SPONSOR/MONITOR'S ACRONYM(S)** N/A |
| | **11. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-RI-RS-TR-2012-252 |

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
This report summarizes the findings of the University of Idaho, Center for Secure and Dependable System's study entitled ``Implications of Multicore Architectures on the Development of Multiple Independent Levels of Security (MILS) Compliant Systems" The purpose of the project is to investigate the security ramifications of various modern microprocessor security architectures in the context of support for MILS compliant systems. This report is divided into three main parts, each of which consists of one or more sections. Part I of this report consists of the introduction and summary of the project. Part II presents a discussion of a framework for evaluating information flow in multicore processors. Part III provides a detailed description of the hypervisor we developed as part of this project to enable experimental evaluation of the Intel processor.

**15. SUBJECT TERMS**
Multi-core processors, Multi-level security, MILS, Multiple Independent Levels of Security

| | | | | | |
|---|---|---|---|---|---|
| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** **WILMAR W. SIFRE** |
| **a. REPORT** U | **b. ABSTRACT** U | **c. THIS PAGE** U | SAR | 239 | **19b. TELEPONE NUMBER (*Include area code*)** **N/A** |

**Standard Form 298 (Rev. 8-98)**
**Prescribed by ANSI Std. Z39.18**

# Abstract

This report summarizes the findings of the University of Idaho, Center for Secure and Dependable System's study entitled "Implications of Multicore Architectures on the Development of Multiple Independent Levels of Security (MILS) Compliant Systems." The purpose of the project is to investigate the security ramifications of various modern microprocessor security architectures in the context of support for MILS compliant systems.

Designers of multicore architectures provide many options for bundling resources in the chip package. Simple architectures put multiple cores on a single chip to share bus interfaces and/or a common cache. System-on-a-Chip (SoC) and System-in-a-Package (SiP) designers choose to integrate additional functionality (e.g., audio and video, encryption engines, analog-to-digital converts, etc.) into the single chip package. As the communication complexity increases between different resources in a chip package, multicore designers continue to look at new methods to increase parallelism and scalability. The Network-on-a-Chip (NoC) method attempts to solve these problems by emulating a modern telecommunications network in a single chip package.

As with any new innovation, security architects and analysts must review the security ramifications of multicore architectures. For example, what new communication channels are present in the multicore architecture and what safeguards are available to protect those communication channels? It is not readily apparent that existing multicore architectures maintain proper information flow isolation so as to enable an implementation of secure systems; but single-core information flow identification and evaluation methods do not scale well to modern multicore architectures.

This report is divided into three main parts, each of which consists of multiple sections. Part I of this report consists of the introduction and summary of the project.

Part II presents a discussion of a framework for evaluating information flow in multicore processors. This framework is the final culmination of this project in that it is an outgrowth of the lessons learned through the analysis we have performed. This report applies the framework to the Cell Broadband Engine Architecture (CBEA), Freescale P4080, and the Intel i7 Nehalem. In our analysis we have found that there are many useful security features in the most recent multicore processors; however, there are known problems and many unknowns that increase the risk when these processors are used in a full multi-level secure environment.

Part III provides a detailed description of the hypervisor we developed as part of this project to enable experimental evaluation of the Intel processor. We found that review of the literature and design documents of the processors analyzed in Part II is insufficient to determine the viability of these processors in an MLS or MILS environment. Additional experimentation is needed to determine the impact of the security features and possible vulnerabilities with those features.

Part IV includes appendices and references.

# TABLE OF CONTENTS

List of Figures ........ vi

List of Tables ........ viii

1 Introduction and Background ........ 2

1.1 Introduction ........ 2

1.2 MILS ........ 2

1.2.1 MILS Principles ........ 3

1.3 The tasks ........ 5

2 Introduction ........ 8

2.1 Why use Multicore Architectures for Secure Systems ........ 9

2.2 Project Objectives ........ 10

2.3 Part II Overview ........ 12

3 Analyzing Multicore and Virtual Machine Architectures ........ 13

3.1 The Basics of Single-core Information Flow Analysis ........ 13

3.2 A Historical Review of Multicore Architectures ........ 15

3.2.1 From Unicore to Multicore Architectures ........ 15

3.3 Virtual Machine Monitor Architectures ........ 20

3.4 Evaluating Multicore and Virtual Machine Vulnerabilities ........ 21

3.5 Concerns about Information Flow Analyses of Multicore and Virtual Machine Architectures ........ 22

4 Defining a Framework for Multicore Information Flow Analysis ........ 23

4.1 A Hypothetical Reference Architecture ........ 23

4.2 Applying the Framework to the Reference Architecture ........ 26

4.2.1 Identify Hardware Components ........ 26

4.2.2 Information Flow Analysis ........ 26

4.2.3 JTAG Debug Port ........ 30

4.3 Framework Benefits, Advantages, and Value ........ 35

5 Revisiting CBEA Using the Framework ........ 37

5.1 Identify Hardware Components ........ 37

5.1.1 Synergistic Processor Element ........ 37

5.1.2 Element Interconnect Bus ........ 41

5.1.3 Cell Broadband Engine Interface Unit ........ 41

5.1.4 Memory Interface Controller ........ 42

5.1.5 Pervasive ........ 42

5.1.6 Hardware Component List ........ 43

5.2 Identify and Evaluate Information Flows ... 44
5.2.1 Synergistic Processor Element ... 45
5.2.2 Element Interconnect Bus ... 49
5.2.3 Cell Broadband Engine Interface Unit ... 50
5.2.4 Memory Interface Controller ... 51
5.2.5 Pervasive ... 52
5.3 Applying the Security Policy ... 53
5.4 Conclusions ... 54
5.5 Framework Enhancements ... 54
6 Evaluating Freescale P4080 using the Framework ... 55
6.1 Identify Hardware Components ... 55
6.1.1 Processor Cores ... 56
6.1.2 CoreNet ... 58
6.1.3 DDR2/DDR3 SDRAM Controller ... 59
6.1.4 Enhanced Local Bus Controller ... 59
6.1.5 High-Speed Peripheral Interface Complex ... 59
6.1.6 Data Path Acceleration Architecture: ... 60
6.1.7 Real Time Debug: ... 68
6.1.8 Hardware Component List ... 69
6.2 Identify and Evaluate Information Flows ... 69
6.2.1 Processor Core ... 69
6.2.2 CoreNet Coherency Fabric ... 85
6.2.3 CoreNet Coherency Cache ... 86
6.2.4 Enhanced Local Bus Controller ... 86
6.2.5 Data Path Acceleration Architecture: Individual Components ... 87
6.2.6 Data Path Acceleration Architecture: Grouped Components ... 89
6.2.7 On-Chip Network: Individual Components ... 90
6.2.8 On-Chip Network: Grouped Components ... 92
6.2.9 Real Time Debug ... 93
6.3 Applying the Security Policy ... 93
6.4 Conclusions ... 94
7 Evaluating Intel Nehalem Architecture Using the Framework ... 95
7.1 Identify Hardware Components ... 96
7.1.1 Processor Cores ... 96

7.2 Identify and Evaluate Information Flows ... 97
7.2.1 Processor Cores ... 97
7.3 Applying the Security Policy ... 101
7.4 Conclusion ... 102
8 Conclusions and Future Work ... 103
8.1 Conclusions ... 103
8.2 Future Work ... 104
8.3 Contributions ... 104
9 Introduction ... 107
9.1 Motivation ... 107
9.2 Research Challenges ... 109
9.3 Contributions ... 110
9.4 Part III Overview ... 111
10 Background ... 112
10.1 MultiCore Architecture Terminology ... 112
10.2 Overview of Multicore Processors Architectures ... 113
10.3 Virtual Machine Monitor ... 115
10.3.1 CPU Virtualization ... 116
10.3.2 Memory Virtualization ... 117
10.3.3 I/O Virtualization ... 119
10.3.4 Interrupt Virtualization ... 120
10.4 Related Work ... 121
10.4.1 Previous Hypervisors ... 121
10.4.2 Security Challenges in Multicore Systems ... 122
10.4.3 Security Architecture Model ... 123
11 Evaluation of Hardware Features for Security in Multicore Systems ... 125
11.1 CPU ... 125
11.1.1 Sensitive but Unprivileged Instructions ... 127
11.2 Memory ... 129
11.2.1 Extended Page Table and Nested Page Table ... 129
11.2.2 Shared Memory ... 130
11.3 I/O Devices ... 131
11.4 External Interrupt ... 132
11.5 Secure Bootup ... 132

11.6 Virtualization of System Resources ..... 133
11.6.1 State of the Logical Processor ..... 133
11.6.2 Virtualization of System Facilities ..... 134
11.7 Hardware Virtualization Technology Summary ..... 134
12 A Lightweight Virtual Machine Monitor for Secure Analysis of Multicore Architectures ..... 136
12.1 Bootstrapping ..... 136
12.1.1 AP Booting Procedure ..... 137
12.2 Operating Modes ..... 138
12.3 Memory Layout ..... 139
12.3.1 Memory Layout in Single VM System ..... 140
12.3.2 Memory Layout in Multiple VM Systems ..... 141
12.4 IAVMM Paging System ..... 143
12.4.1 Host Page Tables ..... 144
12.4.2 Extended Page Table ..... 144
12.5 VM Launching Process ..... 146
12.5.1 VMM Setup ..... 146
12.5.2 Handling of Guest VM Launching ..... 147
12.6 Handling of VM Exits ..... 147
12.7 Summary ..... 148
13 Evaluation Experiments in Multicore Systems ..... 149
13.1 Evaluation of CBEA Processor Architectures ..... 149
13.1.1 A Security Review of the CBEA ..... 149
13.1.2 Evaluation of Shared DRAM System in CBEA ..... 150
13.1.3 Discussion ..... 152
13.2 Evaluation of Intel 64 Hardware Features based on IAVMM ..... 152
13.2.1 Systematic Evaluation ..... 152
13.2.2 Instruction Case Studies ..... 153
13.2.3 Covert Channels Analysis ..... 159
13.2.4 Memory Access Discussion ..... 160
13.2.5 System Registers Analysis ..... 161
13.2.6 VM Exits ..... 162
13.3 Summary ..... 163
14 A Layered Framework for Secure Multicore Architectures ..... 164
14.1 Security Policy Terminology ..... 164

14.2 3-Level Security Framework ........ 165
14.3 Examination of Hardware-Level Security Mechanisms ........ 167
14.3.1 Secure Hardware Mechanisms of Intel VT-x ........ 167
14.4 VMM-Level Security Mechanisms ........ 169
14.5 Verification Experiments from User-level Security Mechanisms ........ 170
14.5.1 Access Control Security ........ 170
14.5.2 Protection Page Tables ........ 170
14.6 Summary ........ 171
15 Formalize Security Policy for Multicore Architectures ........ 172
15.1 Background ........ 172
15.1.1 Layer Assurance Architectures ........ 172
15.1.2 Security Architecture Models ........ 173
15.1.3 Perfect Security Property ........ 174
15.2 Formal Model of Virtual Machine Systems ........ 175
15.2.1 Formal Model of Virtual Machine Systems ........ 175
15.2.2 Event System Model ........ 178
15.3 Example Layered Assurance ........ 181
15.3.1 Hardware Layer ........ 182
15.3.2 VMM Layer ........ 183
15.3.3 User Layer ........ 183
15.3.4 More Specific Examples ........ 184
15.4 Summary ........ 185
16 Part III Conclusion and Future Work ........ 186
16.1 Conclusion ........ 186
16.1.1 Summary of Part III of this project ........ 186
16.2 Future Work ........ 187
17 Bibliography ........ 190
18 Acronym List ........ 200
Appendix A. CBEA SPPORT DATA ........ 206
Appendix B. P4080 Support Data ........ 212
Appendix C. List of Instructions that Cause VM Exits ........ 219
Appendix D. Intel Instruction Summary ........ 221
Appendix E. A List of Terminology Comparison between Intel and AMD ........ 224
Appendix F. A List of Virtualized Resource ........ 225

## LIST OF FIGURES

Figure 1: Secure Multi-Level Communication System, Implemented using Guarded Communication Subsystems ........ 4
Figure 2: Pentium II Block Diagram ........ 15
Figure 3: Pentium D Block Diagram ........ 16
Figure 4: Core 2 Duo Block Diagram ........ 17
Figure 5: CBEA Processor ........ 17
Figure 6: TILE64 Processor ........ 18
Figure 7: Intel Core i7 Processor ........ 19
Figure 8: P4080 Processor ........ 19
Figure 9: Reference Architecture Block Diagram ........ 23
Figure 10: Extended Reference Architecture Block Diagram ........ 31
Figure 11: Reference Architecture Block Diagram for First Security Policy ........ 32
Figure 12: Reference Architecture Block Diagram for Second Security Policy ........ 34
Figure 13: CMOS 90 nm Cell Broadband Engine Block Diagram ........ 38
Figure 14: SXU Block Diagram ........ 39
Figure 15: MFC Block Diagram ........ 40
Figure 16: CBEA spider network ........ 53
Figure 17: P4080 Architecture ........ 55
Figure 18: DPAA Block Diagram ........ 61
Figure 19: DPAA BM Block Diagram ........ 62
Figure 20: DPAA QM Block Diagram ........ 63
Figure 21: DPAA FM Block Diagram ........ 64
Figure 22: DPAA PME Block Diagram ........ 65
Figure 23: DPAA SE Block Diagram ........ 66
Figure 24: Security Mode State Machine ........ 68
Figure 25: Simplified Real Time Debug Block ........ 69
Figure 26: Processor Privilege Levels ........ 96
Figure 27: ICR Register ........ 97
Figure 28: MDR Register Format ........ 99
Figure 29: SIPI Attack ........ 99
Figure 30: Overview of multicore processor architectures ........ 114
Figure 31: Page Translation Mechanisms ........ 119
Figure 32: Interaction between Virtual Machine Monitors and guests ........ 126
Figure 33: State of VMCS and VMX Operation ........ 127
Figure 34: Intel's EPT and AMD's NPT Mechanisms ........ 129
Figure 35: CBE System Memory Map ........ 131
Figure 36: Application Processor Startup Procedure ........ 138
Figure 37: Global Descriptor Table in IAVMM ........ 138
Figure 38: Memory Layout of IAVMM ........ 140
Figure 39: Memory Layout in Multicore Systems ........ 142
Figure 40: Memory Translation from a VMM2 Viewpoint ........ 142
Figure 41: Memory Translation of EPT from a GVM2 viewpoint ........ 143
Figure 42: Linear-Address Translation to a 2 MB page using IA-32e paging (Intel). ........ 144
Figure 43: The procedure to create EPT ........ 145

Figure 44: A DRAM Bank........................................................................................ 151
Figure 45: Access execution time (µs) to shared memory when running multiple SPEs concurrently ................................................................................................................ 151
Figure 46: IAVMM traces CR information while launching a mini_guest .......................... 154
Figure 47: The formats of CR0 and CR4............................................................................ 162
Figure 48: 3-Level Security Policy Framework for Multicore Architectures....................... 166
Figure 49: Assurance architecture of the multicore system................................................. 166
Figure 50: Standard Commuting Diagram for Implementation Correctness........................ 180
Figure 51: Layered Policy and Implementation Models...................................................... 181
Figure 52: Event Procedures of a VM Exit Example .......................................................... 184

## LIST OF TABLES

Table 1: Evaluated Assurance Level Time and Cost ........ 9
Table 2: Reference Multicore Architecture: Registers ........ 24
Table 3: Reference Multicore Architecture: Core Instructions........ 25
Table 4: Reference Multicore Architecture: Registers ........ 25
Table 5: Reference Multicore Architecture: Widget Extension Instructions........ 26
Table 6: Reference Multicore Architecture: Initial Hardware Components........ 26
Table 7: Reference Multicore Architecture: Analysis Results for Communication Bust Timing Information Flows (No flows in this case)........ 27
Table 8: Reference Multicore Architecture: Core Instruction Set Analysis ........ 28
Table 9: Reference Multicore Architecture: Widget Instruction Set Analysis ........ 30
Table 10: Security Policy 1: MMU Mappings........ 32
Table 11: Security Policy 1: IOMMU Mappings ........ 32
Table 12: Security Policy 1: Masked Interrupts ........
Table 13: Security Policy 1: JTAG Pins........ 33
Table 14: Security Policy 2: IOMMU Mappings ........ 34
Table 15: CBEA: Hardware Component List........ 43
Table 16: MMIO Memory Map........ 44
Table 17: SPE Control Instructions ........ 47
Table 18: SPE Channel Instructions ........ 47
Table 19: SPE Compare, Branch and Halt Instructions........ 48
Table 20: MFC SR1 Configuration Options........ 49
Table 21: DPAA Terms and Definitions ........ 60
Table 22: Cryptographic Hardware Accelerators ........ 67
Table 23: P4080: Hardware Component List ........ 70
Table 24: Conditional Branch Control Instructions........ 71
Table 25: Debug Instructions........ 71
Table 26: Memory Synchronization Instructions ........ 72
Table 27: Processor Control Instructions........ 72
Table 28: Load and Store Instructions........ 73
Table 29: Memory Control Instructions ........ 75
Table 30: System Linkage Instructions........ 76
Table 31: No Information Flow or Externally Visible State Change Register Categories..... 76
Table 32: Process Control Registers........ 76
Table 33: Timer Registers........ 78
Table 34: Interrupt Registers ........ 78
Table 35: Guest-mode Mapped Interrupt Registers........ 79
Table 36: Software Use Special-Purpose Registers........ 80
Table 37: Guest-mode Mapped Software Use Special-Purpose Registers ........ 80
Table 38: Core Device Control and Status Register........ 80
Table 39: Internal Debug Registers ........ 81
Table 40: L1 Cache: Special Purpose Registers ........ 81
Table 41: L2 Cache: Special Purpose Registers ........ 82
Table 42: P4080 All IVORs........ 84

Table 43: P4080 Maskable IVORs ........................................ 84
Table 44: DCFG_DEVDISR2 Component Disable Bits ........................................ 91
Table 45: VM-execution Controls for local ASIC Access Protection ........................................ 100
Table 46: Corresponding settings for 26 sensitive but unprivileged instructions which cause VM Exits in Intel VT-x ........................................ 128
Table 47: Summary of HVT Support between Intel VT-x and AMD-V ........................................ 135
Table 48: BIOS-provided available physical RAM E820 ranges (with 4 GB RAM) ........................................ 140
Table 49: Different Flags between host/guest page table entries and EPT entries ........................................ 146
Table 50: Systematic Features of Various VMMs ........................................ 153
Table 51: Selective Exception Bitmap ........................................ 156
Table 52: Protection page table enforced on IAVMM ........................................ 169
Table 53: SPE Memory Load/Store Instructions ........................................ 206
Table 54: SPE Channels ........................................ 207
Table 55: SPE Constant-Formation Instructions ........................................ 208
Table 56: SPE Integer and Logical Instructions ........................................ 208
Table 57: SPE Hint-for-Branch Instructions ........................................ 209
Table 58: SPE Shift and Rotate Instructions ........................................ 210
Table 59: SPE Floating-Point Instructions ........................................ 211
Table 60: Branch and Flow Instructions ........................................ 212
Table 61: Floating-Point Instructions ........................................ 212
Table 62: Floating-Point Status and Control Register Instructions ........................................ 212
Table 63: Floating-Point Arithmetic Instructions ........................................ 213
Table 64: Floating-Point Compare Instructions ........................................ 213
Table 65: Floating-Point Rounding and Conversion Instructions ........................................ 213
Table 66: Simple Integer Instructions ........................................ 214
Table 67: General Purpose Registers ........................................ 215
Table 68: Floating-Point Registers ........................................ 215
Table 69: Branch Registers ........................................ 215
Table 70: Branch Unit Control and Status Register ........................................ 215
Table 71: Hardware Implementation Dependent Register ........................................ 216
Table 72: L1 Cache Registers ........................................ 216
Table 73: L2 Cache Registers ........................................ 216
Table 74: MMU Registers ........................................ 217
Table 75: Performance Monitoring Registers ........................................ 218
Table 76: All the instructions that cause VM Exits ........................................ 219
Table 77: Intel Instruction Extension Summary ........................................ 221
Table 78: Instructions that may Potentially Cause Vulnerabilities ........................................ 222
Table 79: Additional Instructions that may Cause Vulnerabilities ........................................ 223
Table 80: Comparison of Common Terms between Intel VT-x and AMD-V ........................................ 224

# PART I

# INTRODUCTION AND EXECUTIVE SUMMARY

# 1 INTRODUCTION AND BACKGROUND

## 1.1 INTRODUCTION

This report summarizes the findings of the University of Idaho; Center for Secure and Dependable System's study entitled "Implications of Multicore Architectures on the Development of Multiple Independent Levels of Security (MILS) Compliant Systems" The purpose of the project is to investigate the security ramifications of various modern microprocessor architectures in the context of support for MILS compliant systems.

This report is divided into three main parts, each of which consists of multiple sections. Part I of this report consists of the introduction. Part II presents a discussion of a framework for evaluating information flow in multicore processors. This framework is the final culmination of this project in that it is an outgrowth of the lessons learned through the analysis we have performed. It also summarizes the reviews of the processors we have looked at. Part III provides a detailed description of the hypervisor we developed as part of this project to enable experimental evaluation of the Intel processor.

## 1.2 MILS

Modern net-centric concepts are based upon ubiquitous connectivity and standards based services. This is, in fact, the basis upon which the Department of Defense Information Enterprise Architecture is founded. However, to realize the objective of secured availability requires that users and processes with various levels of trust and access share a common infrastructure. The emerging state of the art on this type of Multiple Level Security is based upon the MILS architecture. This architecture consists of a number of high robust components that when combined appropriately can be trusted to enforce two primary principals of Information Assurance.

- Information is only shared with those processes and users allowed by policy.
- Information is not shared with those processes and users disallowed by policy.

In the MILS architecture [AFOTH06, HHOAF05], made popular by the Air Force Research Laboratory High Assurance Middleware for Embedded Systems (HAMES) project, multi-level secure systems are implemented through separation and controlled information flow. The system is built on a foundation consisting of a separation-based infrastructure, the separation kernel (i.e., hypervisor) and secure inter-processor communication (i.e., the Partitioned Communication Service (PCS)). These components isolate individual applications and services and provide the pathways for secure communication. Supporting these pathways are additional components (i.e., guards, cross-domain services, encryption engines, routers, etc.) that implement the system security policy. One approach to the MILS architecture is to develop a Guarded Communication Subsystem (GCS) which is responsible for the "routing" of messages between applications, sending them through the appropriate filters, guards and access decision points.

Consider the exemplary system depicted in Figure 1. In this figure, we have three processors, each hosting a number of processes. Some of the processes (A, B, D, E and F) are applications that have not been fully analyzed with respect to security, and are thus considered untrusted. Assume application A needs to utilize services provided by application F. Requests from A are passed through the MILS Message Router (MMR) [HOL$^+$05,ROAF$^+$06]} which first sends the message through the appropriate guards (e.g., G1 or G2) and then passes it on to the PCS which securely transmits the message to Processor 2 and its MMR. Processor 2 MMR may pass the message through Guard G5 first, and then to the F service engine. Along the way the guards may accept the message, modify it (e.g., add additional metadata, filter contents) or reject it.

The security policy of the system depicted in Figure 1 includes all of the specifications of authorized requests and communication between the untrusted applications. For example, the policy can specify the content and format of requests from A to F (A may be running a Secret level application).

The security policy can be specified as a conjunction of predicates/operations performed on the messages as they travel between the processes. Each guard can be responsible for enforcement of one or more aspect of the policy. The system as a whole enforces the totality of the policy.

### 1.2.1 MILS Principles

The MILS approach to secure system architecture is based on a set of design principles that can be summarized with the following:

*Time and Space Separation*: The MILS architecture requires that the system be architected as a set of functional units, called "partitions" supported by one or more separation mechanisms (e.g., separation kernel, partitioned communication system). Each partition represents a well-defined set of resources and functionality. The MILS separation mechanisms ensure that private resources (e.g., memory, I/O devices) of a partition are kept isolated from other partitions; including residual data in shared resources, hence space separation. In addition, the execution behavior of one partition should not unduly influence the execution of another partition, hence time separation.

*Controlled Information Flow*: The concept of design modularity requires the development of multiple partitions with communication between the partitions. The MILS architecture supports this with the concept of controlled information flow. The MILS separation mechanisms will allow information to flow only along defined communication paths - allowing a controlled exception to full data separation. With this controlled flow, system architects can require that messages be processed by access guards, information re-graders or other security enforcing components.

*Separation Security Policy (TIME)*: The MILS separation mechanisms enforce policies of type-safety, infiltration, mediation and exfiltration. Type safety specifies that the data types of the information flow mechanisms are preserved (e.g., the controlled information flow will not allow overwriting of a bounded buffer). Infiltration specifies that an executing partition is not able to read or otherwise be influenced by private data of another partition (or the separation mechanism). Exfiltration specifies that private data of an executing partition cannot be written to, modify or otherwise influence the private data of another partition. Mediation specifies that an executing partition cannot use private data from one partition to modify or otherwise influence private data of another partition.

*Reference Monitor (NEAT)*: MILS separation mechanisms implement the reference monitor concepts such that they are Non-bypassable, Evaluable, Always invoked and Tamperproof (NEAT). The separation mechanism will always be invoked to control information flow and manage access of private and shared data. In addition, the system is implemented such that there is no other way to provide information flow or access to a partition's data, except through the separation mechanism, hence non-bypassable. To provide high levels of assurance that the system correctly implements the TIME security policy, the system must be designed in a manner that prevents tampering with the separation mechanism and the mechanism must be simple enough to allow for full evaluation.

#### 1.2.1.1 Multicore MILS

If we consider the system in Figure 1 in the context of a multicore processor, we can deploy the processes (guards and applications) to different cores, or we could even map some of the physical processors to cores. In this MILS system, we still need to ensure separation and controlled information flow as depicted in the architectural representation of Figure 1. The open research question is, “Can we securely deploy MILS systems on multicore architectures?” The answer to this question leads to many questions which we address in the specific research tasks outlined below.

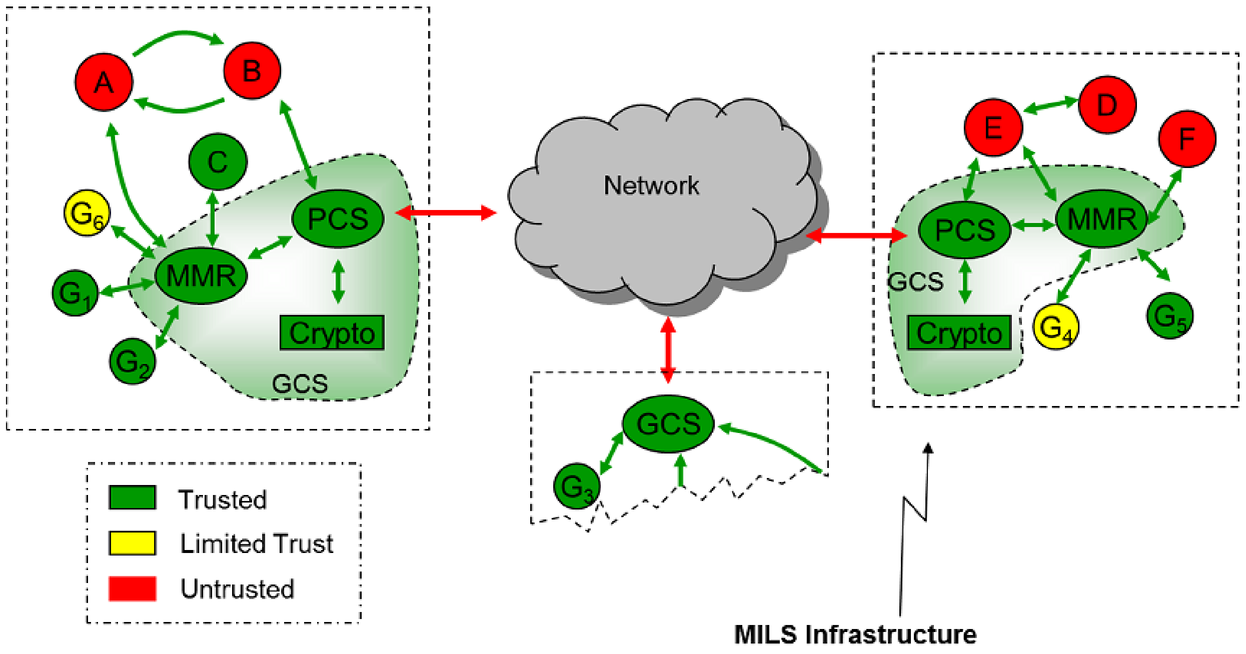


**Figure 1: Secure Multi-Level Communication System, Implemented using Guarded Communication Subsystems**

Commercial multicore architectures provide additional architectural components that facilitate and manage the sharing of information between the cores. For example, the Intel family of processors differentiates between the BootStrap Processor (BSP) and Application Processors

(AP) to enable the BSP to configure and manage the APs after it has configured the rest of the system. In the CBEA processor, the PPC core is the primary core and it controls the activities of the Synergistic Processing Elements (SPE), configuring their memory management units, throughput access to the shared bus, and can start and stop processing on the SPEs. In addition, several multicore processors have implemented internal communication buses (Element Interconnect Bus on CBEA, QuickPath on Intel and CoreNet on Power PC (PPC)) that facilitate transfer of information between cores. Commercial processors also have a variety of memory cache configurations (with combinations of both private shared and partially shared L1, L2 and even L3 caches on chip). In addition to shared memory, management and communication buses, processors such as the CBEA provide a wide range of memory-mapped special purpose registers to facilitate programming. All of these components must be addressed with respect to the MILS security policies.

## 1.3 THE TASKS

The work in this project was divided into six specific tasks. The first three tasks focused on specific microprocessor evaluations while Tasks 4 and 6 focused on recommendations. We found, during the course of this work, that it was best to put the analysis into a framework. Therefore, Part II of this report focuses on the framework and underlying concepts common to this style of analysis, and addresses for Tasks 1 and 3. We found very little difference in the analysis of the AMD and the Intel processors, so we did not focus on that processor in Part I. During the development of the framework we re-examined prior work we had conducted on the CBEA processor, and therefore included that work in this report. As a Power PC based processor, we thought the CBEA would have some similarities with the P4080 (another Power-PC based processor) that would be useful in this analysis. We found some similarities, but many more differences.

Task 5 focused on the development of an experimental hypervisor and use of that hypervisor in analysis of the Intel processor. Some of this work led to results discussed for Tasks 4 and 6.

**Task 1:** Investigate multicore architectural features from representative processors specific to the Intel family of processors, specifically the Intel i-7 processor. The results of this work are presented in Chapter 7, 13 and Appendices C-F.

**Tasks 2.** Investigate multicores architectural features from representative processors specific to the AMD family of processors (instead of the IBM Cell processor as originally planned). There is great similarity between AMD and Intel processor families. The analysis of the Intel processor applies to AMD with the differences highlighted in the Appendix E.

**Task 3.** Investigate multicore architectural features from representative processors specific to the PPC family of processors, specifically the QorIQ family (P4040). The results of this work are presented in Chapter 6 and Appendix B.

**Task 4:** Develop recommendations on the feasibility of utilizing specific architectural components of commercial multicore processors in MILS compliant systems. Part II of this report is the results of this part of the work, with formalizations of the work in Part III Chapters 14 and 15.

**Task 5:** Develop prototype software that allows for the development of experimentation with security functions of the Intel processor investigated in an earlier task. Part III of this report, specifically Chapters 11-13 cover this task.

**Task 6:** Develop software and configuration solutions, mitigations and guidance information regarding selected security concerns uncovered during our investigation. Part II of this report covers this material with formalizations of the work in Part III Chapters 14 and 15.

## 1.4 CONCLUSION

In our analysis, we found that each of the processors we examined presents some security concerns for MILS processes. There is the potential for covert communication channels in any of the processors. In addition, there are so many features, registers and options in these processors, that we cannot be sure that any analysis or checklist is complete. We have found errors, discrepancies and incompleteness in the written documentation, and others have found flaws or undocumented features in the actual processors; all of which can be exploited to break a MILS system. Therefore, there is inherent risk in using any of these microprocessors in MILS or MLS system.

The CBEA, not a specific focus of this work, cannot be used for MILS work without signification additional protections. The synergistic processors can be controlled by the current running process, and can be used as avenues of covert channels.

The Intel and AMD families provide additional protections with their support for virtual machines, which will be essential for MILS systems. However, there are some instructions and modes that allow a VM to impact the rest of the system. Additional experimentation is needed to quantify the possible impact of these instructions and modes to determine the security risk for a MILS implementation on these platforms.

The P4080 is a very complex processor with over 5000 accessible registers, and is really a system-on-a-chip. We found limited documentation on some of these registers (e.g., a register mentioned once, in passing, in documentation about a related feature). Although the system provides security features, we are concerned that configuration of the system is so complex that a comprehensive analysis will not be possible.

# PART II

# FRAMEWORK FOR EVALUATING INFORMATION FLOW SECURITY IN MULTICORE PROCESSORS

# 2 INTRODUCTION

Over the past several decades, the semiconductor industry has effectively leveraged Moore's Law[1] to continually increase the performance of single-threaded processors. Processor designers have taken advantage of the increased transistor density to increase the processor clock frequency and to add additional hardware components in order to improve instruction-level parallelism. Major improvements over these past decades have not been without constraints; signal propagation delay, power consumption, and memory access times now limit the scalability and performance of uniprocessor architectures [SK09]. Moore's Law continues to hold and with the diminishing performance of the uniprocessor architecture the semiconductor industry has started integrating multiple processor cores into a single chip. This chip package is commonly referred to as a multicore processor. This integration has extended the scalability and capabilities of processor architectures in both general purpose and in application-specific markets.

A processor core is an Integrated Circuit (IC) responsible for the reading and execution of program instructions. Initially, the semiconductor process improvements kept pace with the demand for increased throughput by focusing on clock speeds, execution optimizations, and cache sizes. However, reducing the semiconductor process size to add additional gates and improving the clock speed no longer sufficiently meets the large-scale processor demand [MAL08], so manufacturers have started offering multiple microprocessor cores in a single chip package.

Designers of multicore architectures provide many options for bundling resources in the chip package. Simple architectures put multiple cores on a single chip and share bus interfaces and/or a common cache. System-on-a-Chip and System-in-a-Package designers choose to integrate additional functionality (e.g., audio and video, encryption engines, analog-to-digital converters, etc.) into the single chip package. As the communication complexity increases between different resources in a chip package, multicore designers continue to look at new methods to increase parallelism and scalability. The Network-on-a-Chip method attempts to solve these problems by emulating a modern telecommunications network in a single chip package.

Multicore architectures provide an overall performance boost by running multiple independent processes on the separate cores. However, most software applications will not typically see huge performance improvements when run on a multicore architecture. Only well-written, concurrent (i.e., multithreaded) applications will experience the exponential growth potential offered by multicore architectures [Sut05].

As with any new innovation, security architects and analysis must review the security ramifications of multicore architectures. For example, what new communication channels are present in the multicore architecture and what safeguards are available to protect those

[1] Moore's Law states: "The number of transistors incorporated in a chip will approximately double every 24 months" [ine11].

communication channels? This work presents an information flow security analysis of multicore architectures and the effects of multicore architectures in secure systems.

## 2.1 WHY USE MULTICORE ARCHITECTURES FOR SECURE SYSTEMS

The term “secure system” can have multiple meanings. For the purpose of this work, a secure system follows the multilevel security requirements outlined in the Bell and LaPadula security model [BL76]. Simply stated, the Bell-LaPadula security model protects data confidentiality by implementing the *no read-up*[2] and *no write-down*[3] properties. The primary purpose of a secure system is to prohibit data tampering and/or data leakage to users who do not have appropriate clearances. Secure systems typically run in one of three modes: (1) System-High mode, (2) Controlled mode, or (3) Multi-Level Secure (MLS) mode.

System-High mode:

Secure systems running in this mode require that data of different classifications be kept on different computer systems. After this separation, rigorous control of designated clearance levels helps restrict access within the systems [Kar05]. Using the National Security Agency/National Computer Security Center (NSA/NCSC) Rainbow Book Evaluation methodology [Dep87, Dep85a, Dep85b], systems running in System-High mode are typically designated at criteria level B1 and below. The B1 evaluation roughly translates to the Evaluated Assurance Level (EAL) 4 using the Common Criteria Evaluation and Validation scheme, which supersedes the Rainbow Series [Kar05].

Controlled mode:

Secure systems running in this mode require that all users are cleared to some level, but not necessarily the highest level for the information stored on the system [Kar05]. These systems typically evaluate at B2 under the Rainbow Book Evaluation System, which roughly translates to EAL 5 using the newer Common Criteria Evaluation System [Kar05].

Multi-Level Secure mode:

Secure systems running in this mode have information at different classification levels and users who may not be cleared for all the information [Kar05]. These systems typically evaluate at B3 or higher under the Rainbow Book Evaluation System, which roughly translates to EAL 6 or higher using the newer Common Criteria Evaluation System [Kar05]. The evaluation process increases in both cost and time for each assurance level. In 2006, the Government Accountability Office (GAO) published a report showing the time and cost for Common Criteria Product Evaluation [SC06]. A portion of their conclusion appears in Table 1.

**Table 1: Evaluated Assurance Level Time and Cost**

| Assurance Level | Cost ($1000s) | Time (Months) |
|---|---|---|
| EAL 2 | 75 - 200 | 4 - 9 |
| EAL 3 | 110-250 | 7-12 |
| EAL 4 | 150-350 | 9-24 |

[2] A subject is prohibited from reading data from a higher security level.
[3] A subject is prohibited from writing data to a lower security level.

System-High networks became common in the Department of Defense (DoD) [Kar05] due to the falling prices of commodity hardware and the high cost of developing and evaluating more secure systems. One major disadvantage to this approach is that many DoD offices require four or five segregated client systems in order to connect to all of the appropriate networks. To solve the multiple client system problems, NSA began investigating methods for consolidating these multiple System-High mode systems into a single platform. Multicore architectures provide an attractive option for this consolidation by isolating each System-High client in an individual core, with the aim of maintaining separation between every security system embedded within each core.

The most basic requirement of a MLS system is to prevent users who do not have the proper clearance from gaining access to classified information [Kar05]. MILS architectures implement a separation kernel to provide isolated partitions within a multicore architecture. As a result, the MILS solution can be used to implement MLS information systems using multicore architectures.

The University of Idaho Center for Secure and Dependable Systems (CSDS) has a long history of research defining and characterizing MILS-compliant systems [AFOTH06, ROAF+06, HHOAF05]. The work in this report leverages the CSDS information flow analysis work from uniprocessor architectures into a framework analyzing information flow in multicore architectures. While the characteristics of MILS systems are well understood, it is not readily apparent that existing multicore architectures maintain the proper information flow isolation that would enable the implementation of MLS systems in a multicore package.

## 2.2 PROJECT OBJECTIVES

This research is divided into five actionable objectives. The first objective was to leverage research on uniprocessor overt and covert communication channel analysis and experiment with the Cell Broadband Engine Architecture (CBEA) multicore architecture, a Power-PC based multicore architecture. The second objective was to collect the lessons learned from Objective #1 and create a draft comprehensive framework for overt and covert channel analysis for multicore architectures. The third objective was to reanalyze the CBEA using the framework developed in Objective #2. The fourth objective was to analyze the Freescale P4080 multicore architecture using the framework developed in Objective #2. The framework was then revised into its final form as part of this objective. The fifth and last objective was to analyze the Intel Nehalem Architecture with the revised framework developed in Objective #4. More detailed descriptions follow.

Objective 1:

The purpose of the first objective was to research overt and covert channel analysis on uniprocessor systems and to apply those techniques to a specific multicore architecture. This objective required an analysis of different multicore architectures. Initial studies under prior research involved an investigation of the CBEA architecture. This architecture provided a rich diversity of hardware components, which makes a good platform as a baseline for the draft framework that was later developed. As a Power PC based architecture, we felt it also provided a good basis for future Power PC multicore research. The initial results of this research were published in earlier papers [SHAF10a, Smi10, BOAFS10] and are not repeated here.

Objective 2:
The second objective was to review and categorize the research performed on the CBEA in Objective #1. This objective was divided into two tasks. First, a process was defined to create a checklist for all the components needing evaluation in the multicore architecture. Second, a general purpose framework for analyzing each hardware component in the checklist was created. This general purpose component framework satisfied the following requirements:

1. Identify all areas of interest in the component,
2. Provide a description for each area of interest (intended for peer-review),
3. Identify information flows and potential safeguards for each area of interest (this is intended for scalability),
4. Provide an analysis for each area of interest (including any experimental data and code required to reproduce), and
5. Provide the evaluation results for each area of interest.

The result of this objective was a draft framework that provides a complete evaluation for overt and covert communication channels in multicore architectures. This framework is robust enough to allow multiple evaluators to work in parallel, provides enough detail for peer-review and reproducibility, and is robust enough to analyze new or updated hardware components in multicore architectures.

Objective 3:
The purpose of the third objective is to test the draft framework on a known data set. The only known evaluation for overt and covert channels in multicore architectures was published as part of objective #1. The reanalysis of the CBEA will evaluate the effectiveness and identify problem areas in the draft framework. The result of this objective is an organized, peer-reviewable, and comprehensive evaluation of the CBEA. The result of this analysis will be discussed Chapter 5. The draft framework will also be updated to address any deficiencies and/or to simplify or clarify the evaluation process.

Objective 4:
The purpose of the fourth objective is to test the draft framework on a new multicore architecture, the Freescale QorIQ P4 architecture using the P4080 processor. The P4080 processor is targeted towards embedded networking applications (e.g., routers, switches, etc.) and has additional hardware components (e.g., encryption engine, etc.) along with the eight processor cores. The analysis of the P4080 processor using the draft framework will evaluate the effectiveness and identify problem areas in the draft framework. The result of this objective is an organized, peer-reviewable, and comprehensive evaluation of the P4080 processor. The result of this analysis will be discussed in Chapter 6. The framework will be reviewed again to address any deficiencies and/or to simplify or clarify the evaluation process.

Objective 5:
The purpose of the final objective is to test the final framework on the Intel Nehalem Architecture, using the Intel Core i7 processor. The Intel Core i7 is targeted towards general purpose computing. The analysis of the Intel Core i7 will be completed by new multicore

architecture evaluators, to ensure both a usable and a robust final framework. The result of this objective is an organized, peer-reviewable, and comprehensive evaluation of the Intel Core i7 processor. The result shall be broken into at least two sections: (1) evaluation of the core processor and (2) evaluation of processor extensions. The purpose for the two evaluations is to verify the robustness of the framework while they are evaluating architecture updates. The result of these evaluations will be discussed in Chapter 7.

## 2.3 PART II OVERVIEW

Chapter 3 provides background information on multi-level security, virtualization, and multicore architectures. In Chapter 4 we present a framework for analyzing overt and covert channels in multicore architectures. Chapter 5 revisits the Cell Broadband Engine Architecture multicore architecture analysis using the framework presented in Chapter 4. Chapter 6 and Chapter 7 present the evaluation of the Freescale QorIQ P4 Architecture, and the Intel Nehalem Architecture. Finally, conclusions and future work for Part II of this report are presented in Chapter 8.

# 3 ANALYZING MULTICORE AND VIRTUAL MACHINE ARCHITECTURES

Analyzing the information flow security attributes of multicore architectures is not as simple as conducting multiple single-core analyses because most multicore architectures have onboard virtual systems interfacing the application layer with the System-on-a-Chip. This chapter presents an overview of the basic process of information flow security analysis on a single-core system and then shows how that process must be adapted to the complexities of multicore architectures and their onboard virtual machines. A historical review of the development of multicore and virtual machines is provided to show the added complexities of these architectures.

## 3.1 THE BASICS OF SINGLE-CORE INFORMATION FLOW ANALYSIS

One of the many characteristics designers of secure systems must consider is how information can move through the system. Designers use a security policy model to describe the permitted communication channels between the different subjects. In addition to defining the permitted communication channels, secure system designers also need to ensure that communication channels not permitted by the security policy model do not exist. The Trusted Computer Security Evaluation Criteria (TCSEC) Light Pink Book defines a covert channel as:

*Given a nondiscretionary (e.g., mandatory) security policy model M and its interpretation I(M) in an operating system, any potential communication between two subjects I($S_h$) and I($S_i$) of I(M) is covert if and only if any communication between the corresponding subject $S_h$ and $S_i$ of the model M is illegal in M [Dep93, TGC87].*

Covert channels are categorized by the following three criteria:

Type:
Covert channels are typically identified as either a storage or timing communication channel. A storage covert channel exists when the sender subject $S_h$ is able to directly or indirectly alter a storage location where another subject $S_i$ is able to directly or indirectly read this storage location. A timing covert channel exists when the sender subject $S_h$ is able to modulate its use of a resource (e.g., CPU time) and subject $S_i$ is able to directly or indirectly observe this resource modulation.

Noise:
Covert channels are also classified as either noisy or noiseless. A covert channel is classified as noiseless when the communication channel between the sender subject $S_h$ and receiver subject $S_i$ is error free.

Aggregation:
A covert channel where the sender subject $S_h$ uses multiple methods to transmit data to the receiver subject $S_i$. Covert channel aggregation may also occur when a sender subject $S_h$ communicates with multiple receiver subjects.

Evaluating the channel noise and aggregation is important to understand the covert communication channel capacity. The channel capacity is defined as the number of usable bits per second a subject receiver $S_l$ can receive from the subject sender $S_h$. Noisy channels reduce the channel capacity, while aggregation increases the channel capacity. Covert channel capacity is an important attribute of secure system design, given that covert channels with low channel capacity can often be ignored.

The TCSEC Light Pink Book identifies primary methods for analyzing information flow in single-core systems: Syntactic Information-Flow Analysis, Shared Resource Matrix Methodology, and Noninterference Analysis [Dep93].

Syntactic Information-Flow Analysis:
Syntactic Information-Flow Analysis attaches information flow semantics to each statement in the formal specification or to each statement in the implementation [Dep93]. Statements such as X := Y (denoted as $Y \rightarrow X$) cause information to flow from Y to X. Information flows are mapped to the security policy to generate flow formulas. These flow formulas must be proven correct (usually with the help of a theorem prover) to ensure there are no covert storage channels. If the flow formula cannot be proven correct, then further analysis is required to determine if the flow is real or a false positive (i.e., false illegal flow).

Shared Resource Matrix (SRM):
SRM uses a matrix to identify potential covert channels. The SRM analysis method can be applied to formal security policies, informal security policies, and source code. The Trusted Computing Base (TCB) primitives are presented as rows and the TCB variables as columns. The interactions between the TCB primitives and TCB variables are documented in each cell of the SRM, using the following codes:

- R - The TCB primitive can be read.
- M - The TCB primitive can be modified.
- L - The channel is permitted by the security policy.
- N - No useful information can be obtained from the channel.
- S - The sender and receiver is the same process.
- P - A potential channel exists.

Once the direct interactions have been documented in the SRM, a transitive closure operation is performed on the SRM to identify all the indirect interactions.

Non-interference Analysis:
Non-interference Analysis requires the TCB to be viewed as an abstract state machine. Requests to the TCB can be viewed as inputs to the TCB and responses from the TCB can be viewed as outputs. Two subjects $S_h$ and $S_l$ are said to be non-interfering if and only if the output from $S_h$ remains unchanged when all the inputs from $S_l$ are eliminated back to the initial state and when the output from $S_l$ remains unchanged when all the inputs from $S_h$ are eliminated back to the initial state.

## 3.2 A HISTORICAL REVIEW OF MULTICORE ARCHITECTURES

Prior to 2003, the traditional methods for boosting processor performance were to increase the clock frequency, add high-speed, on-chip cache, and optimize instructions [Sut05]. These traditional methods worked for many years until physical issues limited processor clock frequencies around 4 GHz in 2003. Although physical issues limited the processor clock frequency, the transistor size still continued to shrink. Processor manufacturers introduced multicore architectures as the solution to improving processor performance post-2003. Multicore architectures capitalized on smaller process sizes and increased transistor counts to provide multiple processing cores on a single chip.

### 3.2.1 From Unicore to Multicore Architectures

Figure 2 illustrates the high level features of the Intel Pentium II processor (circa 1997). The Central Processing Unit (CPU) core is the heart of the processor where the instructions are executed. The Pentium II processor implements a multi-level on-chip cache as part of the memory hierarchy to address the speed disparity between the CPU core frequency and main memory. The layer 1 cache is segregated into a data cache (L1D) and an instruction cache (L1I).

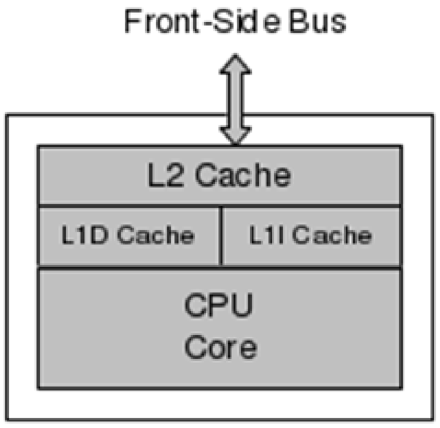


**Figure 2: Pentium II Block Diagram**

In 2002, Intel introduced Intel Hyper-Threading Technology (Intel HT) into the Intel Xeon processor family. Intel HT incorporates Simultaneous Multi-Threading (SMT) into the Intel processor architectures. SMT creates two logical cores that run on a single physical core [MBH+07]. Each logical core maintains an independent processor state, but they share the hardware resources provided by the physical core [MBH+07]. While SMT-enabled processors are not true multicore architectures, they are included in this project because many of the same security concerns and safeguards apply. For example, the Intel HT implementation in the Intel Pentium processor has been shown to be susceptible to covert channel and crypto-analytic side channel attacks [Per05].

In 2005, Intel released the Intel Pentium D processor family and Advanced Micro Devices (AMD) released the Athlon 64 X2 processor family. Both of these processor families are marketed for general purpose desktop computer systems.

The Intel Pentium D processor family glues two Intel Pentium 4 cores together into a single chip. Figure 3 shows the Pentium D block diagram. Each core has its own private L1 and L2 caches. Communication between these two cores uses the off-chip Front-Side Bus (FSB) [PPPC07]. Most processors in the Intel Pentium D processor family did not support Intel HT. The Intel Pentium Processor Extreme Edition 840 is the only processor in this family that supported Intel's Hyper-Threading Technology, providing four logical cores on two physical cores.

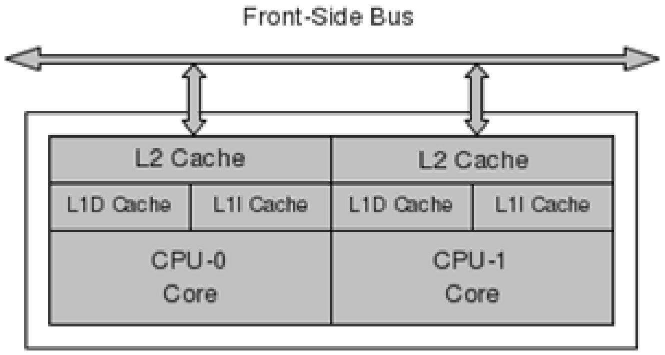


**Figure 3: Pentium D Block Diagram**

The AMD Athlon 64 X2 processor family was designed specifically for multiple cores in a single chip [PPPC07]. Both cores have a private L1 and L2 cache. The AMD Athlon 64 X2 processor family uses a crossbar for connecting both cores to the on-chip memory controller and the Hyper-Transport Bus. This configuration provides inter-core communication at CPU speed by keeping all inter-core communications inside the chip [ine09].

In 2006, Intel released a new multicore architecture family of processors under the Intel Pentium Core brand name. This new family of multicore architectures was intended to replace the Intel Pentium D processor family with an integrated multicore architecture.

Figure 4 illustrates the high-level features of the Intel Core 2 Duo processor. Fundamentally, the Core 2 Duo processor is similar to the traditional single-core processor architecture. The L1 caches and the Data Translation Lookaside Buffer (DTLB) and the Page Miss Handler (PMH) are duplicated for each core. The L2 cache is shared between the two cores.

Also in 2006, a consortium of Sony, Toshiba, and IBM came out with the Cell Broadband Engine Architecture (CBEA) (first available for purchase). Two CBEA based products available for purchase were: Sony PlayStation 3 and the IBM BladeCenter QS20.

The CBEA provides a single general purpose SMT core, making two logical cores. The CBEA also provides specialized cores (typically eight), which are designed for computationally intensive tasks. These specialized cores implement a different instruction set from the general purpose SMT core. Each specialized core also has its own local memory store.

Each CBEA component seen in Figure 5 is connected via four one-way buses configured in a ring. Two of these rings run clock-wise while the other two rings run counter-clock wise. The CBEA uses Direct Memory Access (DMA) to move data between the CBEA components.

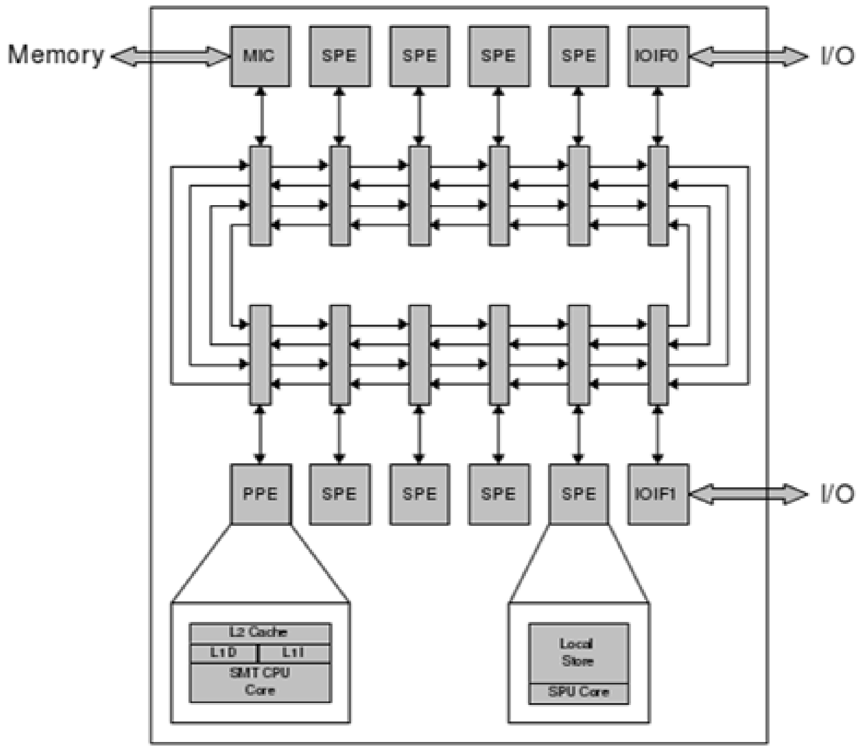


**Figure 4: Core 2 Duo Block Diagram**

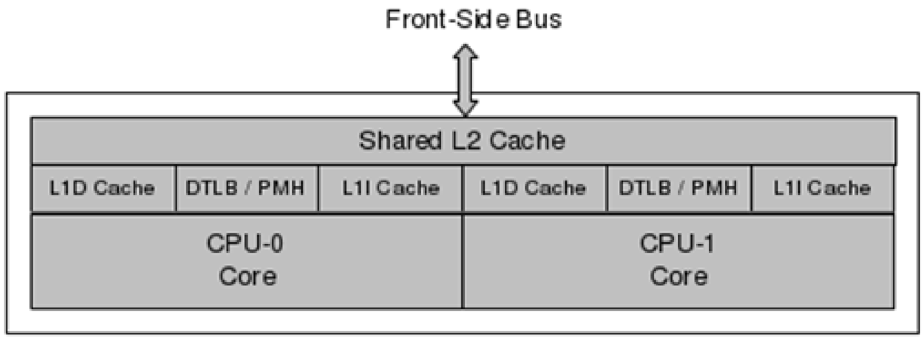


**Figure 5: CBEA Processor**

The CBEA provides a hardware security architecture where one or more of these specialized cores can be put into a secure processing mode [SK09]. When the specialized processing core is in the secure processing mode, the hardware isolates the core so that no other core, operating system or hypervisor can interrogate the internal state of the isolated core [SK09].

In 2007, Tilera released the TILE64 embedded multicore processor. The TILE64 multicore processor contains 64 independent, general purpose cores. Each core has its own private L1 and L2 cache as well as a distributed L3 cache [ine07]. The cores are connected using an intelligent mesh (iMesh), which provides extremely low-latency and high-bandwidth communication between the cores, memory, and other Input/Output (I/O) cache [ine07].

Figure 6 illustrates the high-level features of the Tilera TILE64 processor. Each of the 64 tile processors is a full-featured, general purpose Very Long Instruction Word (VLIW) processor with integrated L1 and L2 caches. The tile processors are connected in a mesh configuration using non-blocking switches. This mesh configuration supports an interconnect bandwidth of 31 Tbps. Each tile processor can run an independent operating system, or the tile processors can be grouped to run a multiprocessing operating system. In addition to the 64 tile processors, this chip also includes memory and I/O controllers.

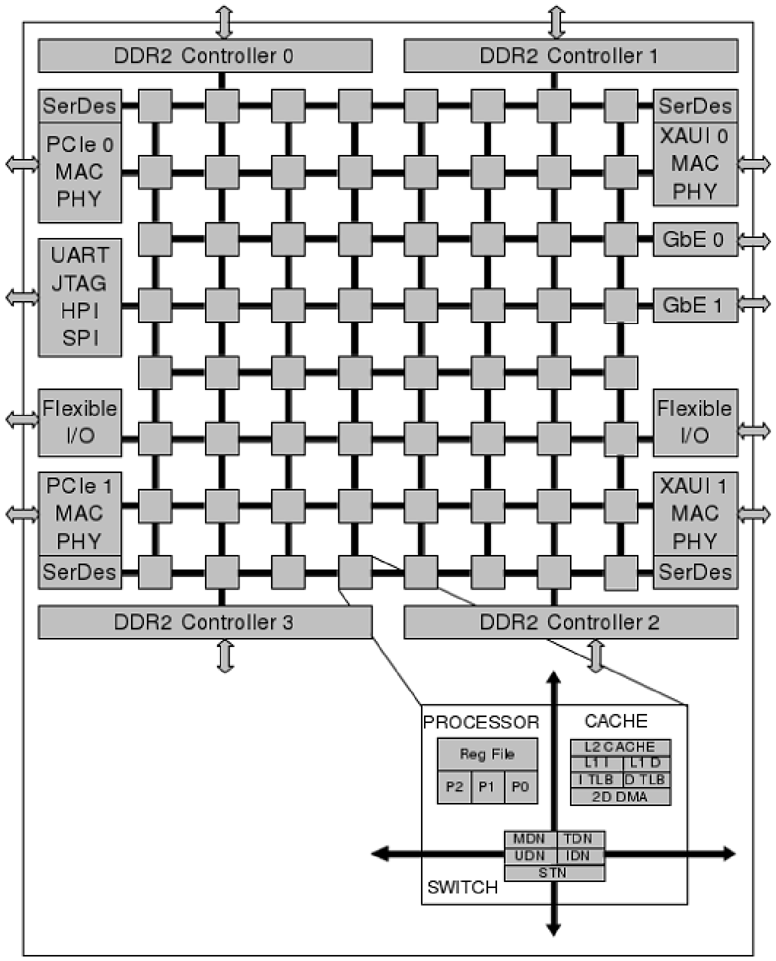


**Figure 6: TILE64 Processor**

In 2008, Intel released a new multicore architecture family of processors under the Intel Core i brand name, see Figure 7. The Intel Core i3 processor brand is targeted towards the budget market, the Intel Core i5 processor family is target for the mid-range market, and the Intel Core i7 processor is targeted towards the high-end processor market. This multicore architecture replaces the Front-Side Bus with the Intel QuickPath Interconnect architecture. This architecture replaces a shared bus architecture with high-speed, packetized, point-to-point communications [pap09].

Also in 2008, Freescale semiconductor also released the QorIQ P4080 Communications Processor. This processor provides eight Power Architecture cores, each with integrated L1 and L2 caches. The QorIQ P4080 Communications Processor also supports a multi-megabyte shared L3 cache. The hardware provides acceleration for encryption, regular expression pattern matching, and Ethernet packet management [ine]. On-chip components are connected by the CoreNet coherency fabric, which manages (1) full cache coherency between the caches and (2) point-to-point, concurrent connectivity between the hardware components. This chip is intended for embedded systems and includes a variety of memory and I/O controllers. Figure 8 illustrates the high-level features of the QorIQ P4080 Communications Processor.

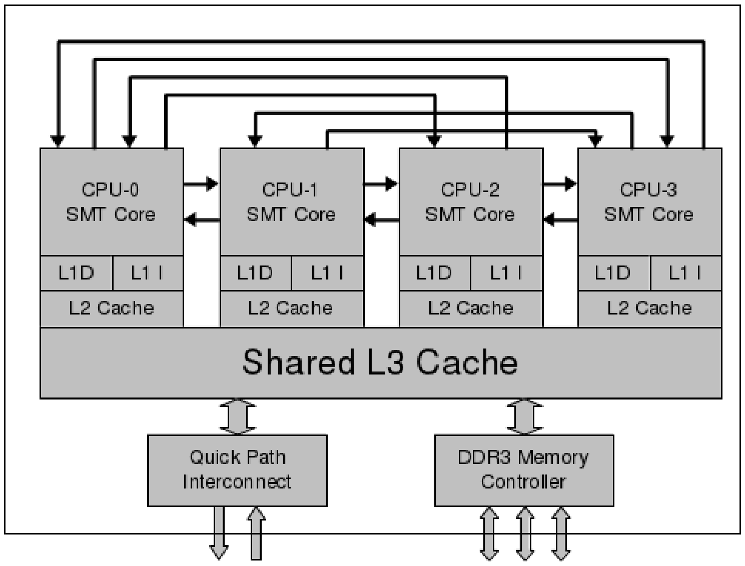


**Figure 7: Intel Core i7 Processor**

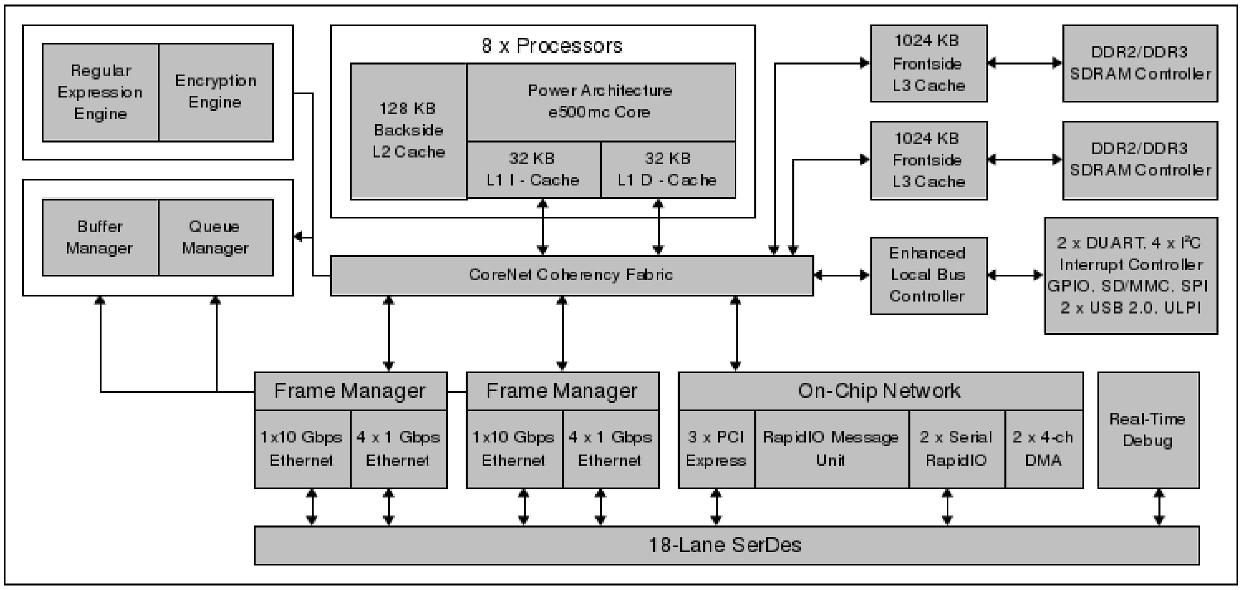


**Figure 8: P4080 Processor**

## 3.3 VIRTUAL MACHINE MONITOR ARCHITECTURES

Modern computer systems are designed as hierarchies of well-defined interfaces. These well-defined interfaces facilitate independent subsystem development by both software and hardware design teams. Typically, these well-defined layers hide the lower-level implementation details in order to reduce design complexity. Virtualization can also take advantage of these well-defined interfaces to present the real system as a different virtual system or as multiple virtual systems [SN05].

There are two prominent types of virtualization used in modern computer systems: (1) Process Virtualization and (2) System Virtualization.

Process Virtualization is a fundamental concept implemented in multiprocessing computer systems such as Unix, Linux, and Windows. In these systems, the memory address space, CPU registers, and other hardware resources are virtualized so each process has the illusion that it is the only process running on the computer system.

System Virtualization requires a hypervisor or Virtual Machine Monitor (VMM) to virtualize the hardware system resources; this will create a virtual environment known as a Virtual Machine (VM) [Dou10]. The Virtual Machine provides the illusion of real hardware resources to software running inside the virtual environment. This report will only focus on System Virtualization. VMMs can be categorized into the following categories:

Classic System VMMs:
Classic System VMMs run on bare metal with the highest privileges. VMs run with lower privileges, allowing the VMM to intercept all VM requests to critical hardware resources. VMWare ESX is an example of a Classic System VMM.

Hosted VMMs:
Hosted VMMs run as an application in the hosting operating system. The VMM relies on the hosting operating system to provide access to the underlying hardware devices. VirtualBox and VMWare Workstation are examples of Hosted VMMs.

Whole System VMMs:
Whole System VMMs provide virtualization for systems that are not necessarily instruction set compatible with the underlying physical hardware. The VMM translates the VM instruction set architecture to the physical hardware instruction set architecture. Whole System VMMs enable software applications that are developed for one system to be run on a completely different system. In addition, Whole System VMMs can also be used to perform binary translations on hardware systems that are not virtualization-friendly. In these cases, the VMM would detect and emulate unsafe VM instructions. QEMU is an example of a Whole System VMM [SN05].

Partitioning VMMs:
Partitioning VMMs divide a larger physical system into smaller virtual systems. There are two partitioning methods: (1) Physical Partitioning and (2) Logical Partitioning. Physically Partitioning VMMs provide a high degree of isolation by dividing the hardware components into

Virtual Machines. Logical partitioning VMMs better utilize the underlying hardware by time-multiplexing the underlying hardware resources.

Codesigned VMMs:
Codesigned VMMs are implemented in hardware for the purpose of translating existing hardware instruction set architectures into proprietary instruction set architectures to improve specific characteristics of the processor (e.g., power, performance, etc.). The Transmeta Crusoe is a well-known example of a Codesigned VMM [SN05].

## 3.4 EVALUATING MULTICORE AND VIRTUAL MACHINE VULNERABILITIES

Goldberg [Gol72] developed a model to determine if a processor architecture would support a Virtual Machine Monitor. Goldberg recognized the development of multiprogramming, and multiprocessing systems were not available to system programmers whose program must run on the bare hardware (i.e., under the operating system). For our framework, the hypervisor (i.e., the VMM) plays a critical role in the security of the multicore architecture. Goldberg's research provided the foundation for the security analysis of the hypervisor.

Robin [Rob99] used Goldberg's research to analyze the Intel Pentium's capability to support a secure VMM. In his analysis, Robin identified 17 instructions that did not meet Goldberg's virtualization requirements. These 17 instructions were sensitive and did not cause a trap in the processor.

Robin also investigated the VMWare product that acted like Goldberg's Type II VMM on the Intel platform. The VMWare product should not have been possible, because of the 17 non-privileged sensitive instructions. Robin determined that VMWare implements a hybrid VMM, which emulates these 17 sensitive instructions in the VMM.

Robin's research provided a guide on how to apply Goldberg's research to uniprocessor systems. His research also provided insight into how VMMs could emulate non-privileged sensitive instructions.

Douglas [Dou10] describes how processor architectures have evolved to provide VMM hardware support. In Douglas' research, he evaluates the Intel, AMD, and Advanced RISC Machines (ARM) TrustZone processors. Douglas provided insight on how this VMM hardware support can be used to protect the VMM.

Suh [Suh05] and Champagne [Cha10] both examined how hardware protections could be applied to VMMs to provide protection from both physical and software attacks. While this report does not address physical attacks, both Suh and Champagnes' research presented insight on specific attack vectors like secure booting and access to main memory, etc.

Son and Alves-Foss [SAF07, SAF06b, SAF06a, SAF09] offer an information flow analysis and covert channel analysis for MLS and MILS system architectures.

In [Fra06] Franz introduces a new Multi-Level Security Virtual Machine architecture (MLS-VM). The MLS-VM architecture allows completely untrusted application programs to perform computations on sensitive data without the risk of leaking secrets. Existing technologies, such as secure boot and a Trusted Platform Module (TPM), ensures the integrity of the MLS-VM. Once MLS-VM is up and running, it provides the appropriate protections for MLS systems.

## 3.5 CONCERNS ABOUT INFORMATION FLOW ANALYSES OF MULTICORE AND VIRTUAL MACHINE ARCHITECTURES

The methods for identifying and evaluating information flows presented in the Light Pink Book were developed when single-core processor architectures were prevalent. During this period, processor architectures evolved by increasing clock frequency, adding cache, and optimizing instructions. The underlying processor architecture remained relatively unchanged. The basics of single-core information flow analysis can be summarized as:

1. Covert Channel Type Analysis
2. Noise
3. Aggregation
4. Syntactic Information-Flow Analysis
5. Shared Resource Matrix Analysis
6. Noninterference Analysis

Multicore architectures represent a fundamental shift in the processor design paradigm. Modern multicore architectures depend on parallelism and integrated hardware components to improve performance. Unicore information flow identification and evaluation methods do not scale well to modern multicore architectures.

University of Idaho researchers investigated the security implications of running MILS-compliant separation kernels on commercially available multicore architectures. Specifically, the University examined the data and the resource sharing components of these multicore architectures to ensure they can be used to ensure type-safety, no infiltration of information between cores, no exfiltration of information between cores, and no mediated information flows across cores. The University began the security review process by identifying the major functional components ala single-core analysis. Using a relatively ad-hoc, but informed, approach to multicore information flow analysis that was based on all the prior literature, the University's effort fell short. Specifically,

1. Some resources were not identified.
2. Some processor instructions were not evaluated (that should have been).
3. Some interrupts were not evaluated (that should have been).
4. Covert channel aggregation was not considered.

Our research asserts that single-core information flow analysis methods are too simplistic to be effectively applied to multicore architectures. A new framework for multicore information flow analysis needs to be designed in such a way as to yield consistent, reproducible, and peer-reviewable results. This project defines that framework in the next chapter.

# 4 DEFINING A FRAMEWORK FOR MULTICORE INFORMATION FLOW ANALYSIS

This chapter defines a framework for multicore information flow analysis. The multicore information flow analysis process consists of three steps:

1. Identify hardware components
2. Identify and evaluate information flows and safeguards with regard to step #1.
3. Apply security policies with regard to step #2

These three steps are flushed out in detail later in this chapter, but first it is necessary to define an example reference multicore architecture upon which the framework can be applied.

## 4.1 A HYPOTHETICAL REFERENCE ARCHITECTURE

This section introduces a hypothetical multicore architecture to be used as a reference for introducing the framework proposed in this research. This reference architecture is intended to be simple enough to not be a burden with excessive details but expressive enough to help future-proof this framework against new developments in multicore architectures.

Figure 9 presents a block diagram of the reference multicore architecture.

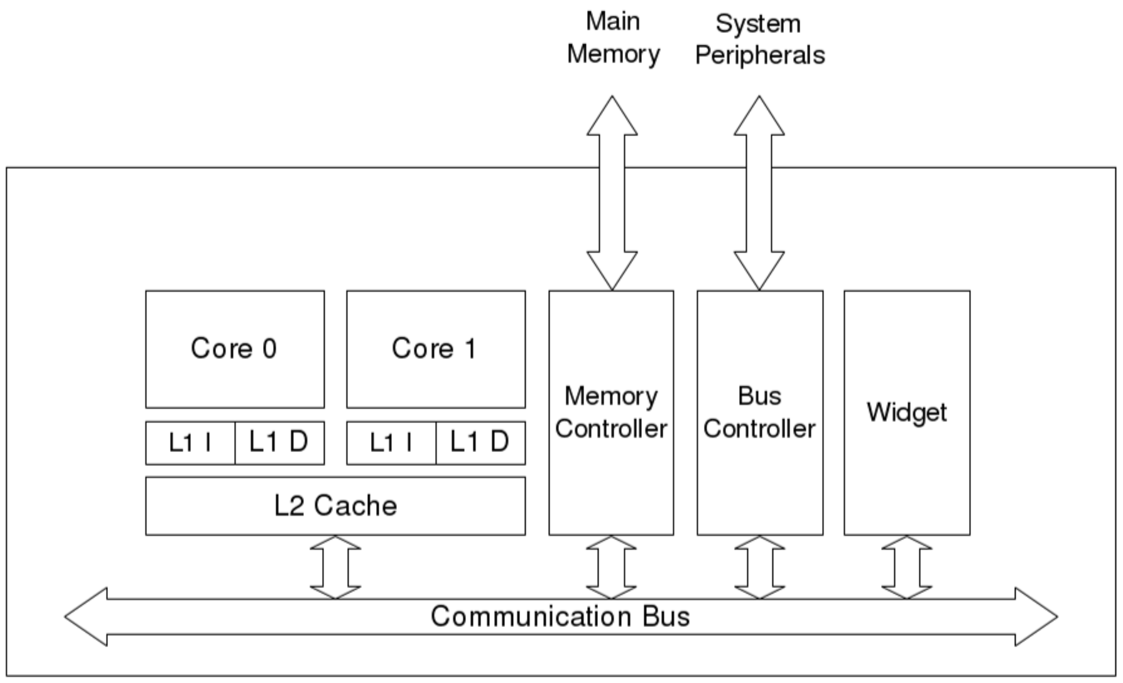


**Figure 9: Reference Architecture Block Diagram**

Communications Bus:

The reference multicore architecture uses the communication bus to communicate between the processor cores, memory controller, bus controller, and the widget. This bus is configured and managed in hardware and does not provide any software controllable configuration. Hardware components are memory mapped into discrete memory address ranges on the communication bus. Hardware prohibits bus snooping by ignoring all bus traffic with destination addresses outside of the memory mapped address ranges.

Only processor cores are permitted to master (i.e., initiate communication on) the communication bus. Bus arbitration is handled by a token. Hardware alternates the token between the two processor cores on a fixed-time schedule. Bus components may only respond to the processor core mastering the bus. Bus transfers must complete before the token rotation; otherwise, hardware delays the bus transfer to the next cycle.

The communication bus also provides a fixed number of interrupt lines. Bus components use these interrupt lines to signal the processing cores when the bus component needs to be serviced.

Processing Cores:

The reference multicore architecture provides two processing cores. Each processing core implements the same register set and instruction set. The processing core provides three hardware protection domains (i.e., privilege levels): Problem, Supervisor, and Hypervisor. The privilege level is the least privileged protection domain and is intended only for applications. The supervisor privilege level is the middle privileged protection domain and is intended for the operating system kernel. The hypervisor privilege level is the highest privilege protection domain and is intended for the hypervisor.

The core set of processor registers are listed in Table 2. The reference architecture provides 32 General Purpose (GP) registers for high-speed access. The Program Counter (PC) register stores the address of the instruction currently being executed. Each processing core can independently be configured to mask (i.e., ignore) interrupts. The INT_MASK register is programmable by the hypervisor to configure how the processor core handles interrupts.

**Table 2: Reference Multicore Architecture: Registers**

| Registers | Description | Privilege |
|---|---|---|
| GP (0 - 31) | 32 General Purpose Registers | Problem |
| PC | Program Counter | Problem |
| INT_MASK | Interrupt Mask | Hypervisor |

The core instruction set is provided in Table 3. Each processing core has a Memory Management Unit (MMU) used to virtualize communication bus addresses to virtual memory addresses. The MMU and Input/Output Memory Management Unit (IOMMU) are programmed by the hypervisor to configure which communication bus addresses are visible to each processing core. Each processing core has a private L1 instruction and L1 data cache. The L2 cache is shared by both processing cores.

**Table 3: Reference Multicore Architecture: Core Instructions**

| Mnemonics | Operand | Description | Operation | Privilege |
|---|---|---|---|---|
| *Arithmetic and Logic Instructions* | | | | |
| ADD | Rt,Ra,Rb | Add | Rt ← Ra + Rb | Problem |
| SUB | Rt,Ra,Rb | Subtract | Rt ← Ra - Rb | Problem |
| MULT | Rt,Ra,Rb | Multiply | Rt ← Ra * Rb | Problem |
| DIV | Rt,Ra,Rb | Divide | Rt ← Ra / Rb | Problem |
| AND | Rt,Ra,Rb | Logical AND | Rt ← Ra & Rb | Problem |
| OR | Rt,Ra,Rb | Logical OR | Rt ← Ra \| Rb | Problem |
| NEG | Rt,Ra | Two's Compliment | Rt ← -Ra | Problem |
| *Branch Instructions* | | | | |
| JMP | A | Jump | PC ← A | Problem |
| BREQ | Ra,Rb,A | Branch if Equal | if(Ra=Rb) then PC ← A | Problem |
| CALL | A | Call Subroutine | R31 ← PC+4, PC ← A | Problem |
| SYSCALL | A | Call Priv Subroutine | R31 ← PC+4, PC ← A | Problem |
| RET | | Subroutine Return | PC ← R31 | Problem |
| *Memory Instructions* | | | | |
| LOAD | Ra,Rb | Load from memory | Ra ← Memory[Rb] | Problem |
| STORE | Ra,Rb | Store in memory | Memory[Ra] ← Rb | Problem |
| MMUCTL | Ra,Rb | Configure the MMU | MMU[Ra] ← Rb | Hypervisor |
| IOMMUCTL | Ra,Rb | Configure the IOMMU | MMU[Ra] ← Rb | Hypervisor |

Memory Controller:

The reference architecture uses the memory controller to map physical memory to an address range on the shared communication bus. The memory control is responsible for translating communication bus addresses to main memory physical addresses. This translation is completed in hardware and is not software configurable.

Bus Controller:

The reference architecture uses the bus controller to communicate with chip external peripherals. The bus controller is responsible for translating communication bus addresses to external device addresses. This translation is performed in hardware and is not software configurable.

Widget:

The widget hardware component is provided in the reference architecture to help future-proof the framework against advancements and changes in multicore architectures. Table 4 lists the widget registers that have been memory mapped to the communication bus. Table 5 lists the extensions to the processing core instruction set to operate the widget component.

**Table 4: Reference Multicore Architecture: Registers**

| Registers | Description | Privilege |
|---|---|---|
| WR (0 - x) | x Widget Registers | Supervisor |

Table 5: Reference Multicore Architecture: Widget Extension Instructions

| Mnemonics | Operand | Description | Operation | Privilege |
|---|---|---|---|---|
| WLOAD | WRa, Ra | Read Widget Register | Ra ← WRa | Supervisor |
| WSTORE | WRa, Ra | Write Widget Register | WRa ← Ra | Supervisor |

## 4.2 APPLYING THE FRAMEWORK TO THE REFERENCE ARCHITECTURE

The remainder of this chapter is devoted to applying this three step analysis to the hypothetical reference multicore architecture defined above.

### 4.2.1 Identify Hardware Components

The first step in the framework is to analyze the multicore architecture to identify and record all the major components. This component list provides a road map during the multicore architecture analysis, potentially identifies under-documented resources, and provides an executive summary of the components analyzed in this report. The executive summary is intended to simplify the evaluation of enhancements to the multicore architecture.

Vendor data sheets typically provide a good resource for initially identifying the major components in the chip package. Hardware interface documentation frequently documents older buses (e.g., Serial Peripheral Interface (SPI)), General Purpose Input/Output (GPIO) pins, and other chip component interfaces.

Using the hardware components identified in Figure 9 and assuming a Joint Test Action Group (JTAG) debug port was identified, reviewing the hardware interface documentation, Table 6 provides the analysis roadmap for the reference multicore architecture.

The evaluated column in Table 6 provides a high-level summary of which components were evaluated during this analysis. This column is intended for a repeated analysis of multicore architectures to evaluate updates and enhancements. For example, many components may not require a reanalysis for the addition of a new instruction set extension.

Table 6: Reference Multicore Architecture: Initial Hardware Components

| Hardware Component | Evaluated |
|---|---|
| Communication Bus | Yes |
| Processor Cores (2) | Yes |
| Memory Controller | Yes |
| Bus Controller | Yes |
| Widget | Yes |
| JTAG Debug Port | Yes |

### 4.2.2 Information Flow Analysis

The next step in the framework is to evaluate the information flows and to identify potential safeguards for each information flow. This step of the framework separates the evaluation of the multicore architecture from the security policy.

#### 4.2.2.1 Communication Bus

The communication bus has two potential information flows: (1) storage information flow and (2) timing information flow.

Storage Information Flow:

The communication bus is configured by hardware instead of software. This configuration eliminates all storage related information flows.

Timing Information Flow:

The communication bus is a shared resource and may be susceptible to timing information flows. Overt bus communications are initiated by a processing core to either load or store data at a specified bus address. Covert timing channels occur when a non-intended receiver is able to observe the sender modulating its use of the bus. The communication bus has the following safeguards to prevent non-aggregated covert timing channels:

- Elimination of bus contention as a shared resource by using a token.
- Elimination of bus usage (i.e. modulation) by rotating the token on a fixed-time schedule.
- Elimination of snooping by enforcing addresses outside of the memory mapped range is ignored.

The characteristics for covert timing information flows can be represented in a table. Each bus component is represented as a table column. Each combination of bus components capable of initiating bus transfers is represented as a table row. Each table cell identifies the covert timing information flow characteristics. The letter "M" indicates the sender is capable of modulating its use of a shared resource. The letter "O" indicates the receiver is capable of observing the modulation of the shared resource. In this particular case, no flows were found.

**Table 7: Reference Multicore Architecture: Analysis Results for Communication Bust Timing Information Flows (No flows in this case)**

| | **Core 0** | **Core 1** | **Memory** | **I/O Bus** | **Widget** |
|---|---|---|---|---|---|
| Core 0 | | | | | |
| Core 1 | | | | | |
| Core 0 + Core 1 | | | | | |

Safeguards:

Table 7 summarizes the lack of covert timing communication channels on the communication bus. No additional safeguards are required for this component to prevent storage or timing covert communication channels.

#### 4.2.2.2 Processor Cores:

Each processor core presents two information flow channels: (1) the communication bus and (2) external interrupts.

Communication Bus Information Flow:

Information flows on the communication bus are generated when processor instructions are executed that have non-local operands. Table 8 provides the analysis of processor instructions to storage locations. Read or Write access to shared storage locations represents potential storage covert communication channels.

**Table 8: Reference Multicore Architecture: Core Instruction Set Analysis**

| | **Privilege Required** | **Non-shared** | | | | **Shared** |
|---|---|---|---|---|---|---|
| | | **GP (0 - 31)** | **PC** | **MMU** | **IOMMU** | **MEM** |
| ADD | Problem | RW | | | | |
| SUB | Problem | RW | | | | |
| MULT | Problem | RW | | | | |
| DIV | Problem | RW | | | | |
| AND | Problem | RW | | | | |
| OR | Problem | RW | | | | |
| NEG | Problem | RW | | | | |
| JMP | Problem | | W | | | |
| BREQ | Problem | R | W | | | |
| CALL | Problem | W | RW | | | |
| SYSCALL | Problem | W | RW | | | |
| RET | Problem | R | W | | | |
| LOAD | Problem | RW | | | | R |
| STORE | Problem | R | | | | W |
| MMUCTL | Hypervisor | R | | W | | |
| IOMMUCTL | Hypervisor | R | | | W | |

External Interrupt Information Flow:

External interrupts provide a potential covert timing communication channel. The receiver could trigger an interrupt request to the sender. The sender could modulate how long it takes to process the interrupt request.

Safeguards:

The hypervisor may configure the MMU and IOMMU to restrict the address space visible to each processing core. The hardware will then prevent the processing core from reading or writing to any address not mapped into the MMU or IOMMU.

The hypervisor manages interrupt signals to each processor core by configuring the INT_MASK register. Masked interrupt signals, however, do not interrupt the processor core.

#### 4.2.2.3 Memory Controller:

The memory controller has two potential information flows: (1) storage information flow and (2) timing information flow.

Storage Information Flow:
The memory controller performs a hardware translation between communication bus addresses and physical memory addresses. The memory controller is not software configurable and offers no direct storage. Physical memory addresses do provide a potential covert storage communication channel.

Timing Information Flow:
The bus infrastructure guarantees that all bus transactions will be completed before the bus token rotates to the other processor core. This configuration eliminates all timing related information flows.

Safeguards:
The memory controller does not provide any direct storage, but the physical memory behind the memory controller does. The memory controller does not provide any safeguards to restrict access to the physical memory. The hypervisor may configure the MMU register in each processing core to restrict access to storage in the physical memory.

#### 4.2.2.4 Bus Controller:

The bus controller has two potential information flows: (1) storage information flow and (2) timing information flow.

Storage Information Flow:
The bus controller performs a hardware translation between communication bus addresses and external peripheral addresses. The bus controller is not software configurable and offers no storage. This configuration eliminates all storage related information flows.

Timing Information Flow:
The bus infrastructure guarantees that all bus transactions will be completed before the bus token rotates to the other processor core. This configuration also prevents the bus controller from acting as the sender and modulating its resources. The bus controller is capable of generating interrupt signals on the communication bus. The interrupt signals provide a potential covert timing information flow by allowing an external peripheral to observe the modulation of a shared resource (i.e., the interrupt signal).

Safeguards:
The bus controller does not provide any direct storage, but external peripherals may provide external storage that could be used as a storage covert channel. The bus controller does not make any safeguards available to restrict access to storage on the peripherals. The hypervisor may configure the IOMMU register in each processing core to restrict access to storage on the external peripherals.

The bus controller does not present any safeguards to prevent potential covert timing channels due to interrupt signals. The hypervisor may configure the INT_MASK register in each processing core to determine which interrupt signals will interrupt the processing core.

#### 4.2.2.5 Widget

The widget has two potential information flows: (1) storage information and (2) timing information flow.

Storage Information Flow:
An extension to the core processor instructions was added to support the widget. Table 9 augments the core instruction set analysis performed in Table 9 with the widget extension instructions.

The widget component introduces a potential storage information flow. The sender (i.e., one of the processing cores) uses the WSTORE instruction to write data to the shared widget registers. The receiver (i.e., the other processing core) uses the WLOAD instruction to read the data from the shared widget registers.

Timing Information Flow:
The bus infrastructure guarantees that all bus transactions will be completed before the bus token rotates to the other processor core. The widget does not generate interrupt signals. The limited instruction set for the widget does not support timing characteristics. This configuration eliminates all timing related information flows.

Safeguards:
The IOMMU on each processor core can be configured by the hypervisor to grant or restrict access to the widget registers.

**Table 9: Reference Multicore Architecture: Widget Instruction Set Analysis**

| | **Privilege Required** | **Non-shared** | | | | **Shared** | |
|---|---|---|---|---|---|---|---|
| | | **GP (0 - 31)** | **PC** | **MMU** | **IOMMU** | **MEM** | **WR (0 - x)** |
| WLOAD | Supervisor | W | | | | | R |
| WSTORE | Supervisor | R | | | | | W |

### 4.2.3 JTAG Debug Port

The JTAG debug port provides access to all storage and timing information inside the reference multicore architecture.

Safeguards:
The JTAG debug port can be disabled by grounding the external JTAG clock line in hardware.

Applying Security Policies
In the previous section, the framework identified the information flows and safeguards. The next step in the framework is to map the information flow analysis and safeguard information to security policies. Separating the information flow analysis from the security policy provides a more scalable and reusable solution. To illustrate this reusability and scalability, the reference multicore architecture (see Figure 10) has been extended to model a system connected to both a Top Secret and Secret network.

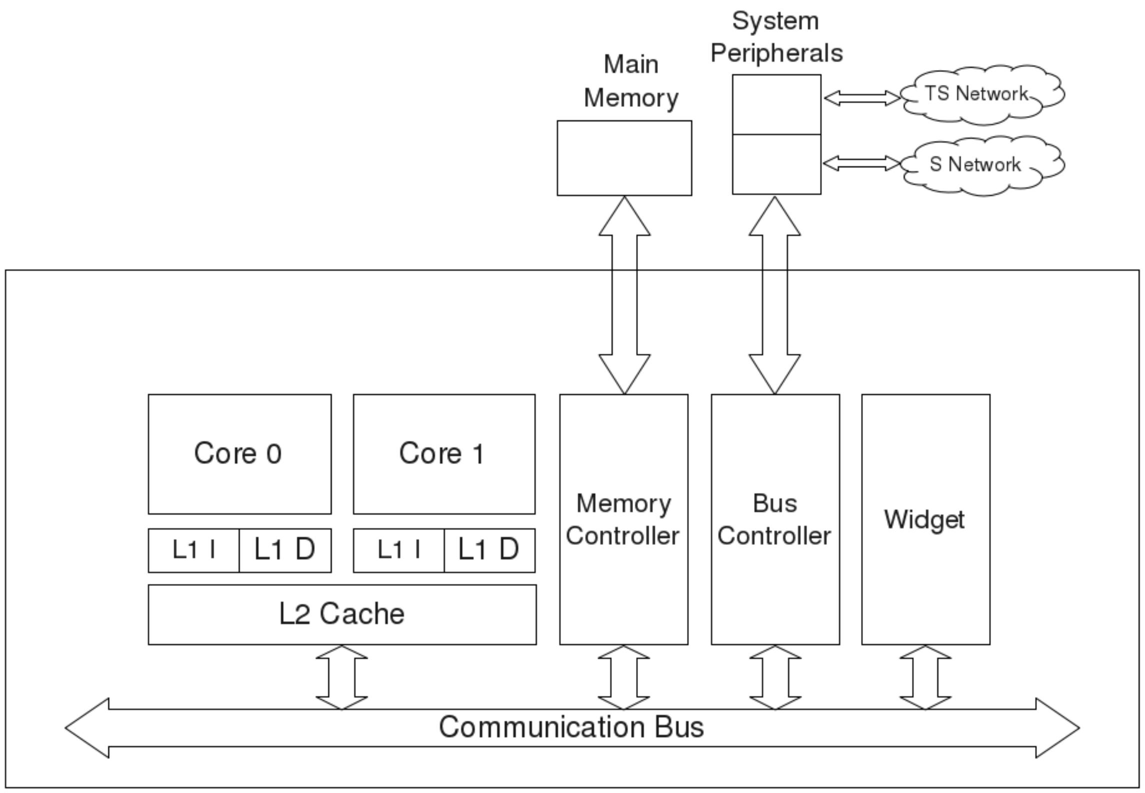


**Figure 10: Extended Reference Architecture Block Diagram**

The extended reference multicore architecture provides two Ethernet controllers as system peripherals. One Ethernet controller is connected to the Top Secret network, while the other Ethernet controller is connected to the Secret network. Each Ethernet controller is then mapped into a distinct address range on the communication bus. Each Ethernet controller is capable of generating external interrupts to signal the processor cores when the data is available.

#### 4.2.3.1 Sample Security Policy 1

The first security policy states:

1. Core 0 - This core shall be run with Top Secret privileges.
2. Core 1 - This core shall be run with Secret privileges.
3. Memory - Memory shall be partitioned into separate Secret and Top Secret regions.
4. External Peripherals - Each Ethernet controller shall be classified at the same clearance level as the network it is connected to.
5. Widget - The widget shall only be accessible at the Top Secret clearance level.

Figure 11 illustrates the extended reference architecture with the first security policy applied.

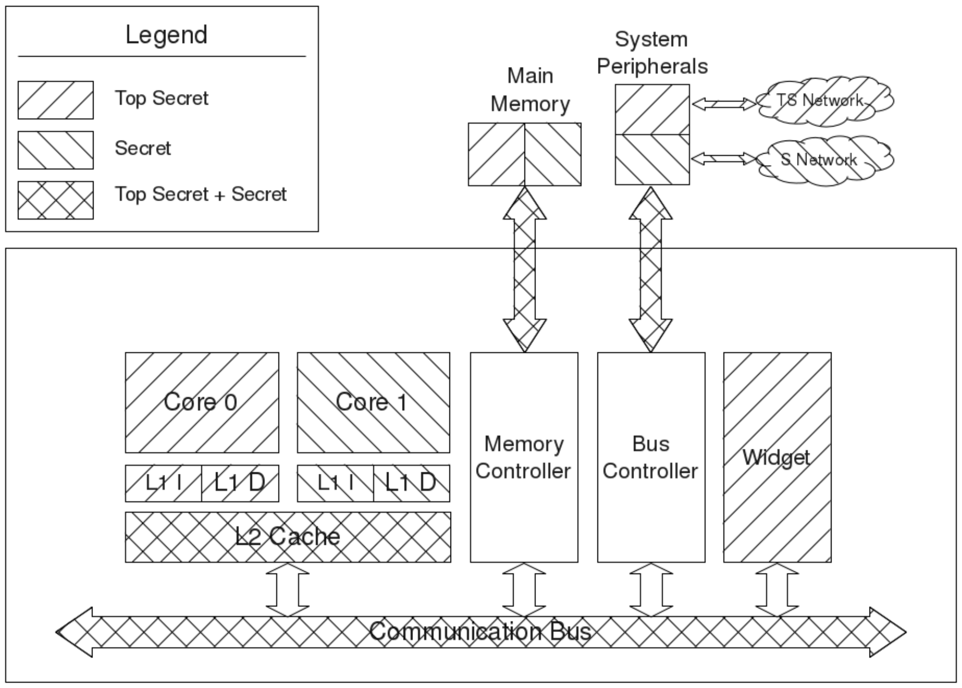


**Figure 11: Reference Architecture Block Diagram for First Security Policy**

The security policy dictates that the physical memory shall be divided into Secret and Top Secret regions. Each processing core shall be permitted to access the memory region if and only if the memory region and the processing core have the same clearance level. Table 10 defines how the hypervisor must configure the visibility of memory addresses to each processor core.

**Table 10: Security Policy 1: MMU Mappings**

| | TS MEM Region | S MEM Region |
|---|---|---|
| Core 0 | Visible | |
| Core 1 | | Visible |

Table 11 defines how the hypervisor must configure the device address range visibility to each processor core. Table 12 defines how the hypervisor must configure interrupt visibility for each processor core. Table 13 defines how the hardware must be configured to disable the JTAG debug port.

**Table 11: Security Policy 1: IOMMU Mappings**

| | Core 0 | Core 1 | TS Ethernet | S Ethernet | Widget |
|---|---|---|---|---|---|
| Core 0 | Visible | | Visible | | Visible |
| Core 1 | | Visible | | Visible | |

**Table 12: Security Policy 1: Masked Interrupts**

| | **TS Ethernet** | **S Ethernet** |
|---|---|---|
| Core 0 | Unmasked | Masked |
| Core 1 | Masked | Unmasked |

**Table 13: Security Policy 1: JTAG Pins**

| | **Pin State** |
|---|---|
| JTAG Clock | Grounded |

#### 4.2.3.2 Sample Security Policy 2

The second security policy states:

1. Core 0 - This core shall be run with Top Secret privileges.
2. Core 1 - This core shall be run with Secret privileges.
3. Memory - Memory shall be partitioned into separate Secret and Top Secret regions.
4. External Peripherals - Each Ethernet controller shall be classified at the same clearance level as the network it is connected to.
5. Widget - The widget shall be accessible to both clearance levels.

Figure 12 illustrates the extended reference architecture with the second security policy applied. The only difference between the two security policies is the security clearance required to access the widget. The hypervisor actions for configuring the MMU (Table 10) masked interrupt signals (Table 12), and JTAG Pins (Table 13) remains unchanged.

The shared widget in the reference architecture presents a problem for this security policy. The WLOAD and WSTORE instructions for the shared widget only require supervisor privilege and do not trap to the hypervisor. The shared widget also introduces a potential covert storage communication channel. The safeguard identified in the framework may not work for this security policy since both processing cores need access to the shared widget.

Three potential solutions exist for implementing this security policy:

1. Select a different multicore processor architecture.
2. Map the required functionality of the shared widget into each processing core without introducing the covert storage communication channel.
3. Use paravirtualization, see Section 10.3.1. Implement the required functionality in the hypervisor and have each processing core make hypervisor calls to implement the shared widget functionality.

This report describes the setup for option #3.

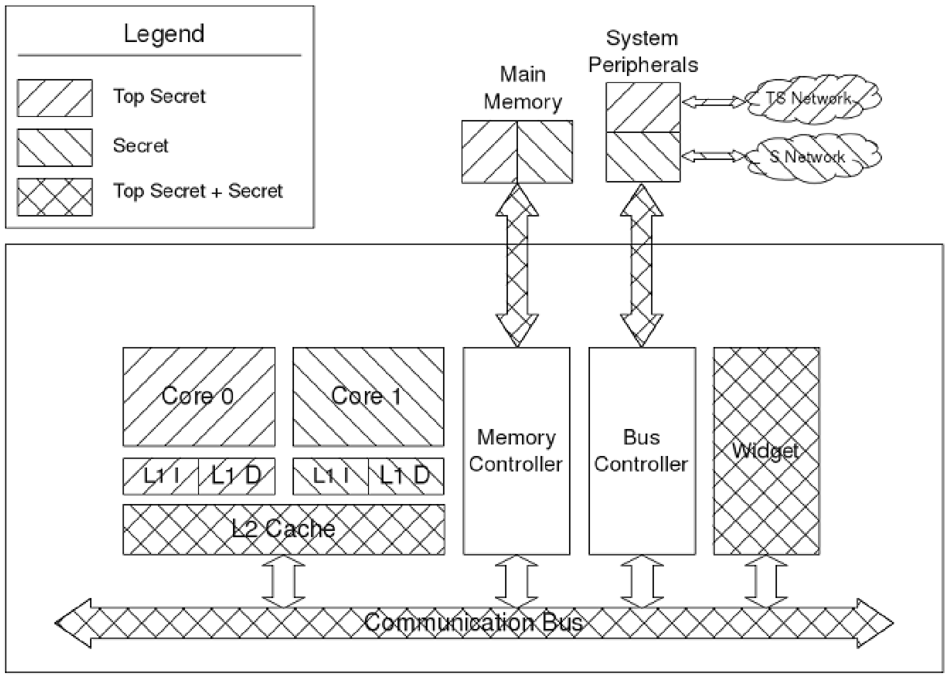


**Figure 12: Reference Architecture Block Diagram for Second Security Policy**

**Table 14: Security Policy 2: IOMMU Mappings**

| | **Core 0** | **Core 1** | **TS Ethernet** | **S Ethernet** | **Widget** |
|---|---|---|---|---|---|
| Core 0 | Visible | | Visible | | |
| Core 1 | | Visible | | Visible | |

Table 14 defines how the hypervisor must configure the device address range visibility to each processor core.

Since the widget device address range is not mapped into the IOMMU of either processor core, WLOAD and WSTORE instructions issued by the processing core will generate page faults. The hypervisor will therefore need to be augmented to provide a hypervisor function call to implement the required widget functionality.

The hypervisor function call would perform the following actions atomically:

1. Enter critical section (i.e., take a mutex to ensure only one processing core is accessing the widget at a time).
2. Map the widget device address range in the processor IOMMU.
3. Perform widget operation.
4. Perform widget cleanup operations.
5. Un-map the widget device address range from the processor IOMMU.
6. Leave the critical section.
7. Exit the hypervisor call.

It is important to note that by not following the recommended safeguard presented in the framework, additional covert communication channels (e.g., a covert timing channel) may be introduced. These potential covert communication channels would need to be reviewed and evaluated independently.

## 4.3 FRAMEWORK BENEFITS, ADVANTAGES, AND VALUE

This chapter introduced a robust and scalable framework for analyzing information flows in multicore architectures. To introduce this framework, first a hypothetical reference multicore architecture was introduced. This reference architecture was simple enough to not overshadow the framework with detail, but expressive enough to ensure that the framework could be applicable to future multicore architectures.

As the framework was applied to the reference multicore architecture, the following information flows and safeguards were identified.

- The design of the communication bus prevented covert storage and covert timing communication channels.
- The processing cores present two potential covert communication channels: Storage channels on shared resources connected to the communication bus and timing channels due to external interrupts. Two safeguards were proposed: (1) Ensure that the hypervisor properly configures the MMU and IOMMU registers to restrict the visible bus address space and (2) Ensure that the hypervisor properly configures the INT_MASK register.
- The design of the memory controller prevented covert storage and covert timing communication channels.
- The bus controller presents a potential covert timing channel with the external interrupts. The bus controller provides no safeguards for preventing this potential communication channel. The INT_MASK register in each processing core provides a potential safeguard.
- The widget provides storage and presents a potential storage covert channel. The processor instructions for reading and writing to the widget registers only require supervisor privilege. The IOMMU registers in the processing core provide a potential safeguard.

The reference architecture was extended to provide a usage scenario with two Ethernet controllers connected to different classified networks. Two security policies were defined using the extended reference architecture. The information flows and safeguards identified in the framework were analyzed and mapped to each of the security policies. The safeguards presented by the reference architecture were sufficient to meet the requirements for the first security policy. The second security policy required augmentation of the hypervisor to support the shared widget resource. Additional analysis for the augmented hypervisor may be required to ensure that additional covert communication channels were not introduced.

This framework provides a scalable, consistent, reproducible, and peer-reviewable solution for analyzing information flows in multicore systems. The framework introduces several scalability improvements:

1. The separation of the hardware flow analysis from the security policies. Hardware flow analysis is coupled to the hardware and only requires updates as hardware changes. Security policies are flexible and change based on need. The hardware flow analysis reuse for multiple security policies reduces secure system evaluation cost and time.
2. The framework provides for incremental updates. Incremental updates are intended for multicore architecture revisions where the changes are localized. For example: As new instruction set extensions are released, only the components touched by the new instructions would require evaluation.

# 5 REVISITING CBEA USING THE FRAMEWORK

The University of Idaho previously evaluated the CBEA for use in a MILS-compliant architecture. This evaluation used a relatively ad-hoc, but informed, approach to multicore information flow analysis. This chapter presents the reanalysis of the CBEA using the framework presented in Chapter 4 as an example of using the framework.

The CBEA does not define a single multicore architecture implementation [Son06]. Instead, the CBEA is intended to provide a flexible architecture to address various system and application requirements. Physically, a CBEA-compliant processor may consist of a single chip or multiple chips. Logically, a CBEA-compliant processor consists of four functional components: the PowerPC Processor Element (PPE), the Synergistic Processor Element (SPE), the Memory Flow Controller (MFC), and the Internal Interrupt Controller (IIC). Every CBEA-compliant processor must provide at least one PPE, at least one SPE, one IIC, and one Element Interconnect Bus (EIB).

## 5.1 IDENTIFY HARDWARE COMPONENTS

The first step in the framework is to identify the hardware components. For this analysis, we analyze the Complementary Metal-Oxide-Semiconductor (CMOS) 90 nm Cell Broadband Engine CBEA-compliant processor. This processor was used in the first and second generation Sony PlayStation 3 devices. Figure 13 provides the block diagram for the CMOS 90 nm Cell Broadband Engine processor [IBM07a].

Figure 13 identifies a number of hardware components that are not capable of working independently. These components are grouped together into logical units that are capable of being evaluated as an independent system-high component.

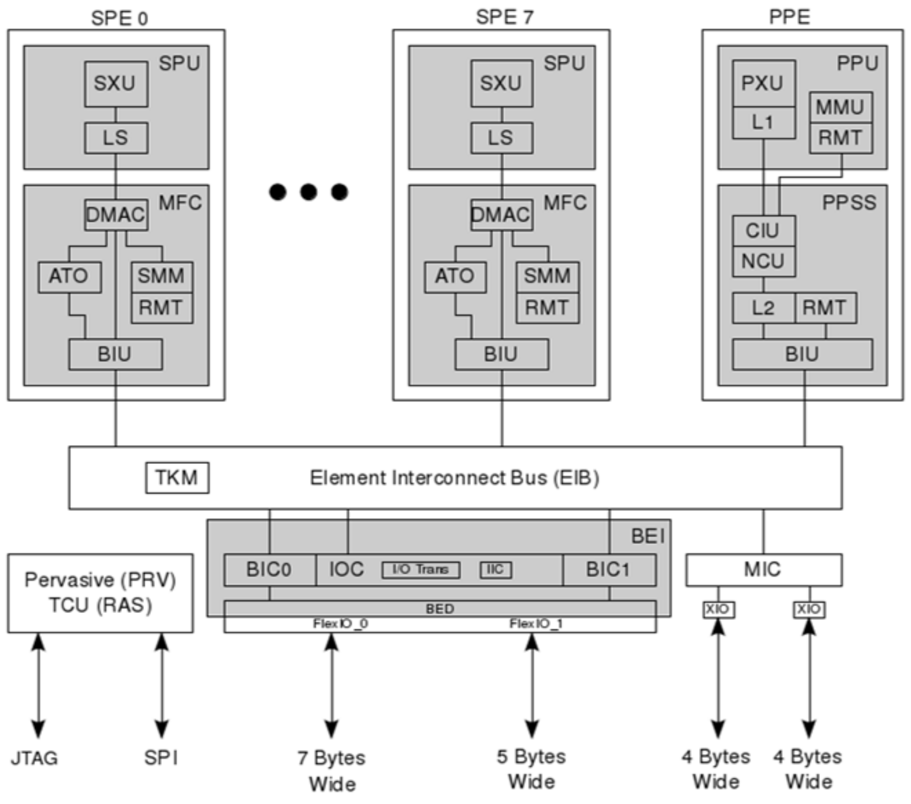
### 5.1.1 Synergistic Processor Element

The synergistic processing element is a unit in the CBEA (see SPE 0 ... SPE 7 in Figure 13) that contains a co-processor (Section 5.1.1.1) and memory flow controller (Section 5.1.1.2).

#### 5.1.1.1 Synergistic Processor Unit

The SPU is logically divided into two hardware components: the Synergistic Execution Unit (SXU) and the Local Store (LS).

Synergistic Execution Unit:
The SXU is the computational unit for the SPU. The SXU is a Reduced Instruction Set Computing (RISC) core that has been optimized for 128-bit Single Instruction, Multiple Data (SIMD) instructions [Sca08, IBM07a]. Figure 14 provides a block diagram of the SXU.



| | | | |
|---|---|---|---|
| ATO | Atomic Unit | MIC | Memory Interface Controller |
| BED | Cell BE Distribution | NCU | Noncacheable Unit |
| BEI | Cell Broadband Engine Interface Unit | PPE | PowerPC Processor Element |
| BIC | Bus Interface Controller | PPSS | PowerPC Processor Storage Subsystem |
| BIU | Bus Interface Unit | PPU | PowerPC Processor Unit |
| CIU | Core Interface Unit | PRV | Pervasive Logic |
| DMAC | Direct Memory Access Controller | PXU | Processor Execution Units (without L1) |
| EIB | Element Interconnect Bus | RAS | Reliability, Availability, Serviceability |
| FlexIO | Frambus Flexible I/O Cell | RMT | Replacement Management Table |
| IIC | Internal Interrupt Controller | SMM | Synergistic Memory Management Unit |
| IOC | I/O Interface Controller | SPE | Synergistic Processor Element |
| I/O Trans | I/O Address Translation | SPI | Serial Peripheral Interface |
| JTAG | IEEE 1149 Test Access Port | SPU | Synergistic Processor Unit |
| L1 | Level 1 Cache | SXU | Synergistic Execution Units (without local store) |
| L2 | Level 2 Cache | TCU | Test Control Unit |
| LS | Local Store | TKM | Token Management Unit |
| MFC | Memory Flow Controller | XIO | Extreme Data Rate (XDR) I/O Cell |

**Figure 13: CMOS 90 nm Cell Broadband Engine Block Diagram**

As shown in Figure 14 the SXU provides six execution units, a SXU register file, and two pipelines. The SXU is capable of executing two instructions concurrently, one instruction on the even pipeline, and the other instruction on the odd pipeline. The SXU register file contains 128, 128-bit general purpose registers and one 128-bit Floating-Point Status and Control Register (FPSCR). The FPSCR register stores result information from floating-point operations. The functionality of the six execution units are described below:

- SPU Control Unit (SCN) - Fetches and dispatches instructions to the other execution units as well as performing the branching and all other control operations
- SPU Even Fixed-Point Unit (SFX) - Handles arithmetic/logic operations and performs comparisons and reciprocation on floating-point values.

- SPU Floating-Point Unit (SFP) - Performs operations on floating-point values; it also multiplies and converts integers
- SPU Odd Fixed Point Unit (SFS) - Shifts quadwords; rotates words, half-words, bytes, and bits; it also shuffles bytes
- SPU Load and Store Unit (SLS) - Performs loads and stores, manages the branch target buffer, and sends DMA requests to the LS
- CPU Channel and DMA Unit (SSC) - Communicates with the MFC and controls DMA data transfer.

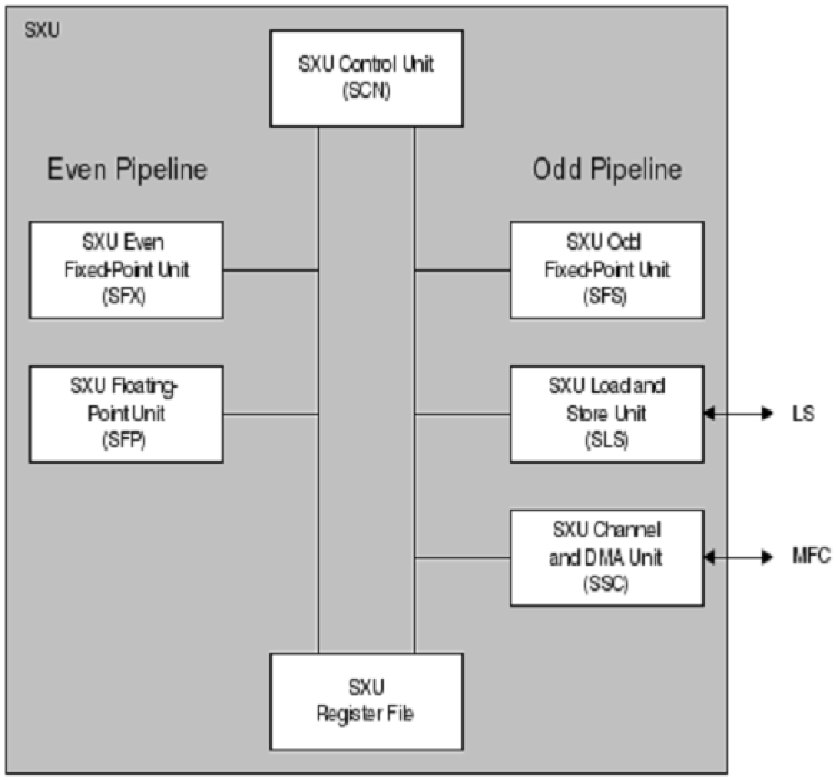

SXU
SXU Control Unit (SCN)
Even Pipeline
Odd Pipeline
SXU Even Fixed-Point Unit (SFX)
SXU Odd Fixed-Point Unit (SFS)
SXU Floating-Point Unit (SFP)
SXU Load and Store Unit (SLS)
LS
SXU Channel and DMA Unit (SSC)
MFC
SXU Register File

**Figure 14: SXU Block Diagram**

<u>Local Store:</u>

This CBEA-compliant processor provides 256 KB of single-ported Static Random-Access Memory (SRAM) local storage for each SPE. The LS is not a cache and does not require tags or a backing store. Since the LS is single-ported, access to the LS is arbitrated using the following priorities [FAD$^+$06]:

1. DMA Transfers
2. SPU Load/Store
3. Instruction Fetch

#### 5.1.1.2 Memory Flow Controller

The MFC provides the interface between the SPU and the EIB. It is responsible for data transfer, protection, and synchronization between the SPU's LS and the main storage domain [Son06]. Figure 15 provides a block diagram for a typical MFC in a CBEA-compliant processor.

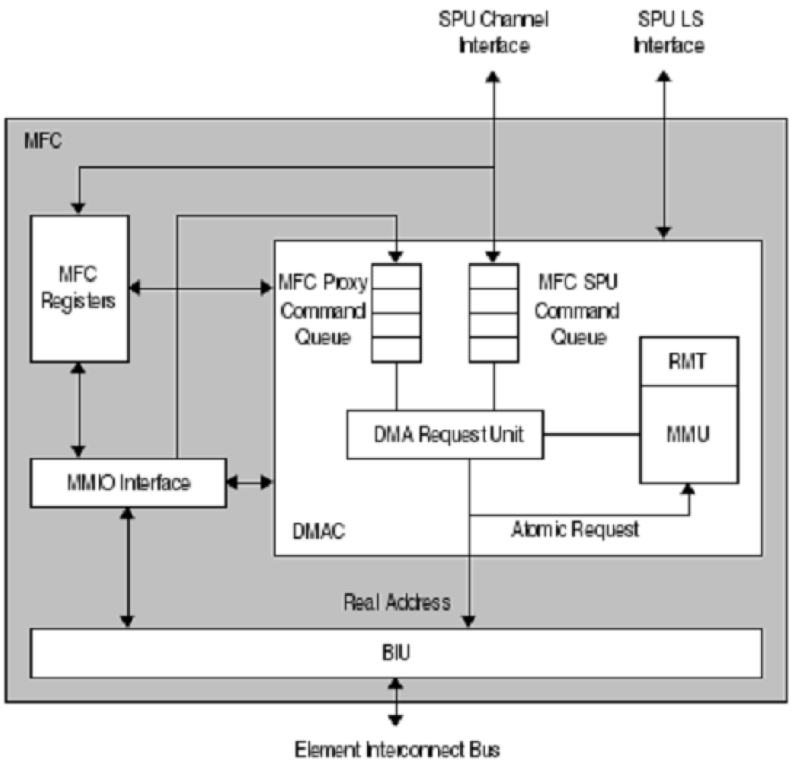


**Figure 15: MFC Block Diagram**

The MFC provides two interfaces to the SPU: the SPU channel interface and the SPU's LS interface [Son06]. The SPU channel interface provides access for the SXU to access MFC facilities and to issue MFC commands. The MFC then uses the SPU's LS interface to access the local storage in the SPU component.

The MFC also provides two interfaces to the Bus Interface Unit (BIU). The first interface provides access for the DMA controller to the EIB. This interface then allows the SPE to communicate with any other component connected to the EIB. The second interface provides Memory Mapped Input/Output (MMIO) access to the MFC facilities. This interface allows other system components to issue MFC commands for the SPE. MFC commands issued using this MMIO interface is called MFC proxy commands.

The MFC provides three primary methods of communication: Direct Memory Access (DMA), Mailboxes, and Signals [Sca08]. DMA is the primary method the MFC uses for transferring large amounts of data. Mailboxes and Signals enable efficient communications for small data sizes. Mailboxes and Signals are messages that are commonly used as DMA control data. For example, Mailboxes may be used to transfer the effective address of the DMA data buffer and Signals may be used to coordinate synchronization between processing units. Fundamentally, the three communication methods all use MFC channels. Each MFC provides 32 channels that are similar to UNIX pipes. Appendix A, Table 54 provides the available MFC channels for each SPE.

### 5.1.2 Element Interconnect Bus

The EIB provides coherent and non-coherent data transfers between connected elements. This CBEA-compliant processor connects the PPE, the eight SPEs, the MIC, and the two FlexIO interfaces using four rings [IBM07a]. Two rings transfer data clockwise and the other two rings transfer data counterclockwise.

To provide lossless communications, the EIB uses a bus arbiter to manage the bus transactions. The arbiter uses command credits to track transactions, prevent collisions, and provide fair access to the bus. An EIB element must have free command credits before it will be granted access to the bus.

### 5.1.3 Cell Broadband Engine Interface Unit

With the exception of the IIC, the Cell Broadband Engine Interface (BEI) component is not specified as part of a CBEA-compliant processor. This means that most of this component is specific to the CMOS 90 nm Cell Broadband Engine processor.

This CBEA processor provides two Rambus FlexIO interfaces for connection to off-chip peripherals [RWW]. Both of these interfaces (IOIF0 and IOIF1) support a non-coherent I/O Interface (IOIF) protocol which is suitable for off-chip I/O peripherals. The IOIF0 interface is software selectable between the non-coherent protocol and the fully coherent EIB protocol. When IOIF0 is configured in coherent mode, the EIB can be extended off-chip. An example use case for this would be to coherently join two CBEA chips in order to produce a cluster.

The BEI provides two primary functions: manages data transfers between the EIB and system peripherals and provides I/O address translation and command processing [RWW].

<u>Bus Interface Controller:</u>
There is not much public information available for the Bus Interface Controller (BIC). The BIC appears to be a coherency bridge between the EIB and the non-coherent system peripherals. This CBEA-compliant processor provides two MMIO regions for each BIC. The BIC NClk MMIO regions do not appear used. The BIC BClk MMIO regions provide a single register to inform the hypervisor when the Rambus FlexIO interface has been initialized.

<u>Input/Output Controller:</u>
The Input/Output Controller (IOC) serves two primary roles: I/O address translation and interrupt handling.

Address translation is handled by a 64-way, 4-set I/O Page Table (IOPT) cache and a 32-way direct mapped I/O Segment Table (IOST).

The IIC is responsible for handling and routing interrupts. The hypervisor can choose to handle the interrupt in the IIC directly, or the IIC can be configured to interrupt the IOC or either of the PPE threads.

### 5.1.4 Memory Interface Controller

The Memory Interface Controller (MIC) provides the interface between the EIB and the main storage [IBM07a]. The MIC supports two Extreme Data Rate (XDR) cell Input/Output (XIO) interfaces, which supports between 64 MB and 64 GB of XDR Random Access Memory (DRAM).

### 5.1.5 Pervasive

The Pervasive component is not specified as part of a CBEA-compliant processor. This means this entire component is implementation specific to the CMOS 90 nm Cell Broadband Engine processor.

This component provides the pervasive performance monitor, power management, thermal management, and the test control unit functionality. This component also provides a Serial Peripheral Interface (SPI) for the Power-On-Reset (POR) initialization sequence. A Joint Test Action Group (JTAG) interface is also provided for low-level debugging and testing purposes.

Performance Monitor:
The performance monitor has the potential to gather data from every component. When enabled, this performance data is stored in trace buffers. At least part of this trace data is available via a MMIO region called the Trace Logic Array (TLA). It does not appear from the publicly available documentation that the performance monitor would cause an information flow back to the other components. Additional experiments on this CBEA-compliant processor with full hypervisor access would be needed to verify this.

Power Management:
The power management provides the ability to alter the divisor for the CBE core clock frequency. Altering the divisor for the CBE core clock provides an externally visible state change.

Thermal Management:
The thermal management monitors thermal conditions for the processor. When a thermal condition is detected two actions may occur: (1) a thermal interrupt is generated to the PPE and/or (2) execution cores (PPE and SPE) may be throttled or stopped.

The external interrupt to the PPE provides information flow if a component is able to modify its thermal output. The throttling or stopping the execution of a processing core could provide an externally visible state change if a component is able to modify its thermal output. Additional experiments on this CBEA-compliant processor with full hypervisor access would be needed to determine if components are able to modify their thermal output.

The thermal management provides the *Thermal Sensor Interrupt Mask Register* to prevent interrupt status bits from generating a thermal management interrupt to the PPE.

Test Control Unit:
This CBEA-compliant processor provides several Fault Isolation Registers (FIRs) for collecting and reporting information about errors [IBM07a]. These FIRs are organized into two categories:

global FIRs and local FIRs. Local FIRs are the lowest level of error collection and are located in different components in the processor. Error status from the local FIRs are ORed together and reported in the global FIRs. The global FIRs are located in the Test Control Unit.

#### 5.1.5.1 Serial Peripheral Interface

This CBEA-compliant processor is initialized in two sequences: POR initialization sequence and firmware initialization sequence [IBM07a]. The POR initialization sequence requires assistance from an external system controller. The external system controller initializes and calibrates specific hardware components in the CBEA-compliant processor using the SPI interface. Once the POR initialization sequence is complete, the system reset interrupt is fired, which starts the PPE executing in hypervisor mode to complete the firmware initialization sequence.

The external system controller is the master for the SPI bus. This configuration allows the external system controller to read and/or modify data inside the processor. While the external system controller is not in scope for this evaluation, it is important to identify and address potential information flows that the external system controller may introduce via the SPI bus.

The SPI interface to the CBEA-compliant processor does provide a SPI Signal *SPI_EN* pin which is used to enable or disable the SPI interface. Since the external system controller is required for POR initialization the *SPI_EN* pin cannot be permanently disabled in hardware. Also, since the external system controller is the SPI bus master, the CBEA-compliant processor provides no additional safeguards to ensure that the *SPI_EN* pin remains disabled after the POR sequence has completed.

#### 5.1.5.2 Joint Test Action Group

This CBEA-compliant processor provides a JTAG interface. Unfortunately, this JTAG interface was not described in any of the publicly available documentation for this processor. It is fair to assume that the JTAG interface would provide significant information flows. As a result, hardware experimentation is needed to determine if this JTAG interface could be disabled.

### 5.1.6 Hardware Component List

Table 15 provides a list of the independent, system-high components evaluated for this chapter. The initial evaluation of the CBEA was split between the University of Idaho and a customer. The customer evaluated the PPE hardware component, while the University of Idaho evaluated the non-PPE hardware components. The framework handles this split evaluation effort by identifying which components are evaluated in the report. Table 15 reflects the work performed by the University of Idaho and the component evaluations provided by this report.

**Table 15: CBEA: Hardware Component List**

| Hardware Component | Evaluated |
|---|---|
| PPE | No |
| SPEs (8) | Yes |
| EIB | Yes |
| BEI | Yes |
| MIC | Yes |
| Pervasive | Yes |

## 5.2 IDENTIFY AND EVALUATE INFORMATION FLOWS

The second step in the framework is to identify and evaluate the information flows between the hardware components identified in the first step.

The PowerPC architecture supports three privilege states: Hypervisor State[4] Privilege State[5], and Problem State [IBM07a]. Trusted software, running on the PPE, configures the HV and PR bits in the *Machine State Register* (MSR) to set the processor state. The analysis of the PPE and the trusted hypervisor is not part of this report.

Table 16 provides the list of MMIO regions[6] [IBM07a]. For this information flow analysis, this report assumes all memory regions marked "Yes" in the "HV" column are properly configured to only be accessible by the trusted hypervisor.

Most of the components identified in Table 15 provide trace and debug information. This report assumes that this functionality will be disabled in production systems and ignores these information flows in areas where hypervisor privilege is required.

**Table 16: MMIO Memory Map**

| Start | End | HV | Area |
|---|---|---|---|
| 0x000000 | 0x3FFFFF | No | SPE Local Store, Problem State, and Privilege 2 |
| 0x040000 | 0x40FFFF | Yes | SPE Privilege 1 |
| 0x500000 | 0x500FFF | Yes | PPE Privilege |
| 0x501000 | 0x507FFF | Yes | Reserved |
| 0x508000 | 0x508FFF | Yes | IIC |
| 0x509000 | 0x5093FF | Yes | Reserved |
| 0x509400 | 0x5097FF | Yes | Pervasive: Performance Monitor |
| 0x509800 | 0x509BFF | Yes | Pervasive: Thermal and Power Management |
| 0x509C00 | 0x509FFF | Yes | Pervasive: RAS |
| 0x50A000 | 0x50AFFF | Yes | MIC and TKM |
| 0x50B000 | 0x50FFFF | Yes | Reserved |
| 0x510000 | 0x510FFF | Yes | IOC I/O Address Translation |
| 0x511000 | 0x5113FF | Yes | BIC 0 NClk |
| 0x511400 | 0x5117FF | Yes | BIC 1 NClk |
| 0x511800 | 0x511BFF | Yes | EIB |
| 0x511C00 | 0x511FFF | Yes | IOC I/O Command |
| 0x512000 | 0x512FFF | Yes | BIC 0 BClk |
| 0x513000 | 0x513FFF | Yes | BIC 1 BClk |
| 0x514000 | 0x514FFF | Yes | Reserved |
| 0x515000 | 0x7FFFFF | Yes | Reserved |

[4] This is also called Privilege 1 State.

[5] This is also called Privilege 2 State.

[6] This MMIO region starts at the address specified in the BP_Base register. The BP_Base register is initialized as part of the Power-On-Reset sequence.

### 5.2.1 Synergistic Processor Element

This section provides the analysis of the SPE described in Section 5.1.1.

#### 5.2.1.1 Information Flows

A major design goal for the CBEA was to replace Application Specific Integrated Circuit (ASIC) components with general purpose, high-speed data processing cores [SK09]. Due to this design goal, the SPEs were not designed to support general purpose operating systems. Instead the SPEs were optimized for data-rich operations assigned to them by the PPE [IBM]. The CBEA-compliant processor under evaluation provides eight SPE cores. In this report, each SPE is evaluated as an independent system-high compartment.

Another design goal for the CBEA was to address memory latencies introduced by slower memory technology and the memory hierarchy [SK09]. To overcome this memory latency problem, the SPE is logically divided into two components: the SPU and the MFC. The SPU is not able to access the main storage directly. The SPU is only able to access SPU private memory (i.e., the Local Store). The MFC is responsible for transferring data between the SPU LS and the main storage. This design allows for main storage data accesses to be processed concurrently with SPU instruction processing.

To identify information flows, we logically wrap the SPE component in a fully-enclosed polyhedron metaphor. The surface color of the polyhedron represents the externally visible state; the mapping of colors to state is defined by the security analyst, determining the specific security relevant states. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

<u>MFC Channel Interface:</u>
The MFC provides the channel interface to facilitate data transfer between itself and other system components. The channel interface produces information flows breaching the polyhedron surface. Several of the channels in this interface are blocking channels, which alter the externally visible state.

<u>MFC MMIO Interface:</u>
The MFC also provides an MMIO interface allowing other system components access to the internal MFC facilities. This MMIO interface produces information flows breaching the polyhedron surface.

<u>Interrupts:</u>
The MFC generates three classes of interrupts: Class 0 (Error), Class 1 (DMA Translation), and Class 2 (Application).

<u>SXU Instruction Analysis:</u>
The SXU instruction set provides 199 instructions, categorized into nine categories.

The Memory Load/Store instructions (Appendix A, Table 53) only resolve addresses in the SPU LS. Since the SXU and the LS are both in the same system-high compartment, these instructions do not generate information flows breaching the polyhedron surface. Also, the execution of any of these instructions will not alter the externally visible state of the SPE system-high compartment.

The Constant-Formation instructions (Appendix A, Table 54) only manipulate data stored in the SXU general purpose registers. The execution of these instructions will not generate information flows breaching the polyhedron surface or altering the externally visible state of the SPE system-high compartment.

The Integer and Logical instructions (Appendix A, Table 56) only manipulate data stored in the SXU general purpose registers. The SXU instruction set does not support instructions that cause processor exceptions (e.g., division instructions could potentially generate a division by zero exception). With no outside data accesses and no processor exceptions possible, these instructions will not generate information flows breaching the polyhedron surface or alter the externally visible state of the SPE system-high compartment.

The Hint-for-Branch instructions (Appendix A, Table 57) only provide a hint for upcoming branches. These instructions do not alter the current state of the SXU, but instead are designed to help optimize the pre-fetching of instructions for the SPU LS. Since the SXU and the LS are both in the same system-high compartment, these instructions do not generate information flows that breach the polyhedron surface or alter the externally visible state of the SPE system-high compartment.

The Control instructions (Table 17) need additional investigation. The **stop** and **stopd** instructions are commonly used to signal the PPE for assistance (e.g., the SPE application has exited, or the SPE requests a PPE assisted callback). These instructions pass information to the PPE, thus causing an information flow to breach the polyhedron surface. These instructions also alter the externally visible state by stopping the execution of instructions in the SXU. The **lnop** and **nop** instructions do not cause information flows or externally visible state changes. The **sync** and **dsync** instruction stalls the execution of instructions until earlier loads, stores, and channel instructions have completed. The loads and the stores are to the SPU LS and would not cause an information flow. An experiment is needed to determine if stalling the SXU for channel instructions could be used as a covert channel. It is unclear from the publicly available documentation which special purpose registers the **mtspr** and **mfspr** instructions operate on. Additional documentation and/or experiments are needed to determine if these instructions generate information flows that breach the polyhedron surface and/or alter the externally visible state.

The Shift and Rotate instructions (Appendix A, Table 58) only manipulate data stored in the SXU general purpose registers. The execution of these instructions will not generate information flows that breach the polyhedron surface or alter the externally visible state of the SPE system-high compartment.

**Table 17: SPE Control Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Stop and Signal | stop | Yes | 1.0 |
| Stop and Signal with Dependencies | stopd | Yes | 1.0 |
| No Operation (Load) | lnop | Yes | 1.0 |
| No Operation (Execute) | nop | Yes | 1.0 |
| Synchronize | sync | Yes | 1.0 |
| Synchronize Data | dsync | Yes | 1.0 |
| Move from Special-Purpose Register | mfspr | Yes | 1.0 |
| Move to Special-Purpose Register | mtspr | Yes | 1.0 |

The Channel instructions (Table 18) require additional scrutiny. The **rdch** and **wrch** instructions breach the polyhedron surface by transferring data between system-high compartments. An experiment is needed to determine if the **rchcnt** instruction could be used for a covert communication channel. Specifically, is it possible for a different system-high compartment running at a different clearance level to modulate the channel capacity?

**Table 18: SPE Channel Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Read Channel | rdch | Yes | 1.0 |
| Read Channel Count | rchcnt | Yes | 1.0 |
| Write Channel | wrch | Yes | 1.0 |

The Compare, Branch, and Halt instructions (Table 19) also require additional scrutiny. The compare instructions do not generate information flows that breach the polyhedron or alter the externally visible state of the SPE system-high compartment. These branch instructions (**bi**, **iret**, **bisled**, **bisl**, **biz**, **binz**, **bihz**, and **bihnz)** provide software control over the interrupt processing state. When executing those branch instructions, the software chooses to enable interrupt processing, disable interrupt processing, or leave interrupt processing unmodified. The modification of the interrupt processing for the SPE is considered an externally visible state change due to the fact that it may provide a potential covert timing channel. The Halt instructions cause the SXU processor to halt when the test condition is true. Halting the SXU causes an externally visible state change given that it may be possible to either directly detect or infer when the SPE processor is not executing instructions.

The Floating-Point instructions (Appendix A, Table 59) only manipulate data stored in the SXU general purpose registers and the FPSCR register. The SXU float-point architecture does not support processor exceptions; instead the FPSCR register is updated and instruction execution continues. It is the responsibility of the software to check the FPSCR for errors. The execution of these instructions will not generate information flows breaching the polyhedron surface or alter the externally visible state of the SPE system-high compartment.

### 5.2.1.2 Safeguards

MFC Channel Interface:

The MFC DMA transfers are from the SPE point of view. The DMA **get** command retrieves data from an external address and the DMA **put** command pushes data to an external address. In either direction, the external address must be accessible to the SPE. The PPE provides an MMU and each SPE provides an MMU. These MMUs must be programmed properly to ensure that undesired information flows are prohibited.

**Table 19: SPE Compare, Branch and Halt Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Halt If Equal | heq | Yes | 1.0 |
| Halt If Equal Immediate | heqi | Yes | 1.0 |
| Halt If Greater Than | hgt | Yes | 1.0 |
| Halt If Greater Than Immediate | hgti | Yes | 1.0 |
| Halt If Logically Greater Than | hlgt | Yes | 1.0 |
| Halt If Logically Greater Than Immediate | hlgti | Yes | 1.0 |
| Compare Equal Byte | ceqb | Yes | 1.0 |
| Compare Equal Byte Immediate | ceqbi | Yes | 1.0 |
| Compare Equal Halfword | ceqh | Yes | 1.0 |
| Compare Equal Halfword Immediate | ceqhi | Yes | 1.0 |
| Compare Equal Word | ceq | Yes | 1.0 |
| Compare Equal Word Immediate | ceqi | Yes | 1.0 |
| Compare Greater Than Byte | cgtb | Yes | 1.0 |
| Compare Greater Than Byte Immediate | cgtbi | Yes | 1.0 |
| Compare Greater Than Halfword | cgth | Yes | 1.0 |
| Compare Greater Than Halfword Immediate | cgthi | Yes | 1.0 |
| Compare Greater Than Word | cgt | Yes | 1.0 |
| Compare Greater Than Word Immediate | cgti | Yes | 1.0 |
| Compare Logical Greater Than Byte | clgtb | Yes | 1.0 |
| Compare Logical Greater Than Byte Immediate | clgtbi | Yes | 1.0 |
| Compare Logical Greater Than Halfword | clgth | Yes | 1.0 |
| Compare Logical Greater Than Halfword Immediate | clghti | Yes | 1.0 |
| Compare Logical Greater Than Word | clgt | Yes | 1.0 |
| Compare Logical Greater Than Word Immediate | clgti | Yes | 1.0 |
| Branch Relative | br | Yes | 1.0 |
| Branch Absolute | bra | Yes | 1.0 |
| Branch Relative and Set Link | brsl | Yes | 1.0 |
| Branch Absolute and Set Link | brasl | Yes | 1.0 |
| Branch Indirect | bi | Yes | 1.0 |
| Interrupt Return | iret | Yes | 1.0 |
| Branch Indirect and Set Link if External Data | bisled | Yes | 1.0 |
| Branch Indirect and Set Link | bisl | Yes | 1.0 |
| Branch If Not Zero Word | brnz | Yes | 1.0 |
| Branch If Zero Word | brz | Yes | 1.0 |
| Branch If Not Zero Halfword | brhnz | Yes | 1.0 |
| Branch If Zero Halfword | brhz | Yes | 1.0 |

|  |  |  |  |
|---|---|---|---|
| Branch Indirect If Zero | biz | Yes | 1.0 |
| Branch Indirect If Not Zero | binz | Yes | 1.0 |
| Branch Indirect If Zero Halfword | bihz | Yes | 1.0 |
| Branch Indirect If Not Zero Halfword | bihnz | Yes | 1.0 |

MFC MMIO Interface:
The MFC MMIO interface is accessed via effective memory addresses. The PPE and SPEs provide MMUs. These MMUs must be programmed properly to ensure that undesired information flows are prohibited.

**Note:** this architecture was designed for the PPE processor to manage the SPE cores. This means the PPE will most likely require access to the each MFC MMIO interface.

Interrupts:
Each MFC provides the *Interrupt Registers* for managing interrupts [Son06]. These registers are designated as Privilege 1 and they are intended for the hypervisor to configure and control. The INT_Route register provides interrupt priority and destination for each class of interrupts. The *INT_Mask_class0*, *INT_Mask_class1*, and *INT_Mask_class2* registers provide control over which interrupts are permitted to generate external interrupts.

SXU Instructions:
The information flows (including interrupts) can be managed with the previously mentioned safeguards. The CBEA-compliant processor does not provide any additional safeguards to protect against externally visible states.

SPE Registers:
Each MFC provides the *MFC_SR1* register to provide high-level, programmable control over the MFC configuration. This register is designated as Privilege 1 and is intended for the hypervisor to configure and control. Table 20 describes the MFC configuration options available via this register.

**Table 20: MFC SR1 Configuration Options**

| Field Name | Brief Description |
|---|---|
| TL | Software or Hardware page table search |
| S | SPU master run control |
| R | Enable effective address translation |
| PR | Restrict MFC access to problem state pages |
| T | Honor broadcast TLB Invalidate instructions |
| D | Map LS into the system address space |

### 5.2.2 Element Interconnect Bus

This section provides the information flow analysis of the EIB described in Section 5.1.2.

#### 5.2.2.1 Information Flows

To identify information flows, we logically wrap the EIB component in a fully-enclosed polyhedron. The surface color of the polyhedron represents the externally visible state; the mapping of color to state is defined by the analyst. Using this framework:

1. Any information flow that breaches any surface of the polyhedron must be evaluated.
2. Any information flow that modifies the externally visible state must be evaluated.
3. Any information flow that does not breach the polyhedron surface or modify the externally visible state can be ignored.

<u>Data Transfer:</u>
The primary purpose of the EIB is to transfer data between EIB elements. The data breaches two of the polyhedron surfaces: once as the data enters the EIB and again when the data leaves the EIB.

<u>Livelock Interrupt:</u>
The EIB provides internal logic to detect a livelock. The EIB can be configured to generate an interrupt when the livelock condition is detected. The interrupt then causes an information flow that breaches the surface of the polyhedron.

<u>Capacity:</u>
The EIB manages fairness using a command credit system. A potential covert channel may exist if EIB elements with different classifications are able to observe and modulate the use of command credits. Additional experiments are needed to determine if this potential covert channel exists.

#### 5.2.2.2 Safeguards

<u>Data Transfer:</u>
The EIB does not provide many methods for restricting information flow on the EIB bus. The only solution the EIB provides is to disable command credits for the EIB element, which in essence locks the element out of the EIB.

<u>Livelock Interrupt:</u>
Anytime the Livelock condition is detected, the *LLD_Int* bit will be updated in the *EIB_CFG* MMIO register. An interrupt will be generated when the *LLD_Mask* bit in the *EIB_AC0_CTL* MMIO register is set to 1. To disable the interrupt (and the information flows) set the *LLD_Mask* bit to 0.

<u>Capacity:</u>
MMIO registers in the EIB and TKM MMIO regions provide configuration options for allocating command credits. If experimentation identifies a potential covert channel, these configuration options could be tuned to potentially eliminate the covert channel. Additional experimentation would be required.

### 5.2.3 Cell Broadband Engine Interface Unit

This section provides the information flow analysis of the BEI described in Section 5.1.3.

#### 5.2.3.1 Information Flows

To identify information flows, we logically wrap the BEI component in a fully-enclosed polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow that breaches any surface of the polyhedron must be evaluated.
2. Any information flow that modifies the externally visible state must be evaluated.
3. Any information flow that does not breach the polyhedron surface or modify the externally visible state can be ignored.

Data Transfer:
The primary purpose of the BIC is to transfer data between EIB elements and the system peripherals. This data transfer breaches two of the polyhedron surfaces: once as the data enters the BIC and again when the data leaves the BIC.

Address Translation Cache:
A potential information flow may exist by exploiting the address translation cache. Additional experiments are needed to determine if the address translation cache could provide a covert communication channel.

Capacity:
The analysis of off-chip peripherals is not in scope for this analysis. The security policy analysis should still consider the potential covert channels due to bandwidth capacity for off-chip busses.

Interrupts:
An information flow exists if the IIC is configured to route interrupts to either of the PPE threads.

#### 5.2.3.2 Safeguards

Data Transfer:
The BEI provides the IO*C_BaseAddr0*, *IOC_BaseAddrMask0*, *IOC_BaseAddr1*, and *IOC_BaseAddrMask1* registers for BIC address translation. These address translation registers must be programmed properly.

Also, the PPE and SPEs provide MMUs, which must be programmed properly to ensure that undesired information flows are prohibited.

Interrupts:
The *IIC_IR* register in the IIC provides software configuration for the priority and routing of interrupts.

### 5.2.4 Memory Interface Controller

This section provides the information flow analysis of the MIC described in Section 5.1.4.

#### 5.2.4.1 Information Flows

To identify information flows, we logically wrap the MIC component in a fully-enclosed polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow that breaches any surface of the polyhedron must be evaluated.
2. Any information flow that modifies the externally visible state must be evaluated.
3. Any information flow that does not breach the polyhedron surface or modify the externally visible state can be ignored.

Data Transfer:

The primary purpose of the MIC is to transfer data between EIB elements and main storage. The data breaches two of the polyhedron surfaces: once as the data enters the MIC and again when the data leaves the MIC.

Capacity:

The MIC provides a specific memory and channel bandwidth to the EIB. Depending on the hardware configuration, (i.e., the number of banks of memory, memory configuration, etc.) a timing information flow may exist if the cooperating EIB elements are able to modulate the consumption of the memory and/or EIB channel bandwidth. Additional experimentation is needed to identify the existence of this covert communication channel.

#### 5.2.4.2 Safeguards

Data Transfer:

The MIC relies on the proper configuration of the MMUs in the PPE and the SPEs to restrict access to the main memory store.

Capacity:

The memory channel capacity is subject to hardware constraints. Managing EIB command credits may provide a method for eliminating this potential channel. Additional experimentation is required to verify any safeguard properly addresses the covert channel potential.

### 5.2.5 Pervasive

This section provides the information flow analysis of the pervasive unit described in Section 5.1.5.

#### 5.2.5.1 Information Flows

The component is assumed to be under complete control from the trusted hypervisor. No information flow analysis was performed.

#### 5.2.5.2 Safeguards

Trace and debug capabilities should be disabled. Also, the trusted hypervisor should ensure that all the MMIO regions specified in Table 16 are properly protected.

## 5.3 APPLYING THE SECURITY POLICY

The final step in the framework is to apply the Security Policy to the information flows identified in the previous step.

Initially, the University of Idaho evaluated the CBEA architecture for use as a generic MILS capable multicore architecture. The University of Idaho looked at using the CBEA architecture as a security gateway to bridge red and black networks. Red networks are cleared to transmit classified and/or sensitive information in plaintext. Black networks are cleared to transmit encrypted classified and/or sensitive information. CBEA spider network illustrates how two security gateways would be used to transmit classified and/or sensitive information between two red networks via a black network.

In this scenario, a Sony PlayStation 3 is used for the CBEA-compliant processor. The CBEA compliant processors are used as the red-black network security gateways. Each security gateway will use four SPE processors. Three SPE processors are used to encrypt red network data or decrypt black network data. The fourth SPE processor serves as a guard to verify that any data transmitted across the black network is encrypted.

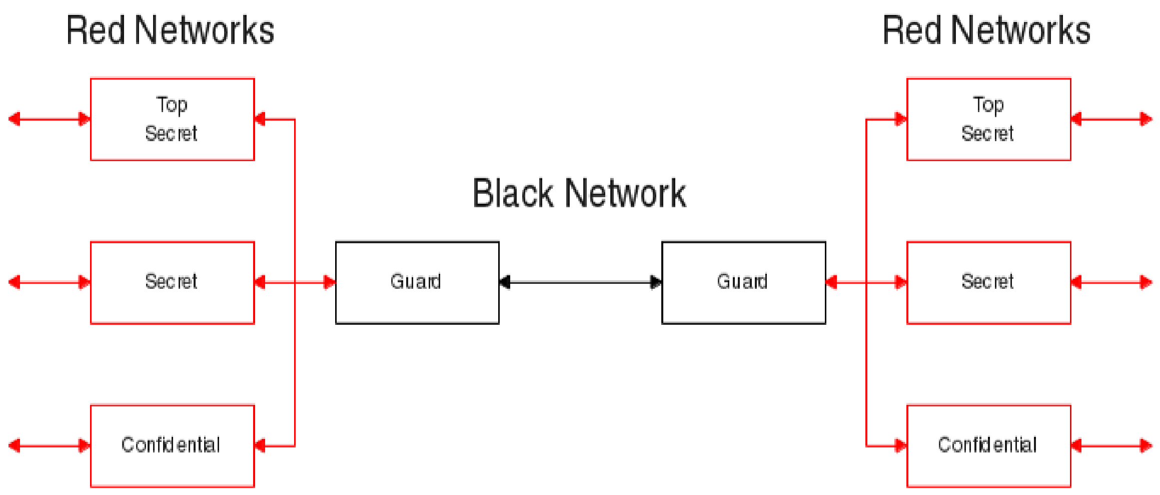


**Figure 16: CBEA spider network**

The security policy used for this report:

1. Red network data is only permitted to be transmitted to a red network with the same classification. Separate cryptographic keys shall be used to ensure the separation of data at different classification levels.
2. All encrypted data must pass through the guard before transmission on the black network.

## 5.4 CONCLUSIONS

For this report, we reanalyzed the information flows and safeguards against the security policy for the red-black network security bridge[7]. Based on the partial analysis provided in the previous step, we cannot recommend the CBEA multicore architecture as a generic MILS architecture for the following reasons:

- SPE cores are not intended for general purpose processing.
- The most complex component (i.e., the PPE) must be trusted. Also, the PPE must run as a MLS compartment instead of a system-high component to securely manage the SPE cores.
- Blocking MFC communication channels may provide covert timing communication channels.

## 5.5 FRAMEWORK ENHANCEMENTS

During the analysis the following enhancements were made:

1. The six-sided box metaphor was changed to a fully-enclosed polyhedron. While the six-sided box metaphor accomplished the goal of providing a fully enclosed area, there is nothing special about six sides. The fully-enclosed polyhedron accomplishes the same goal without the restriction of six sides.
2. The original framework did not take into consideration the externally visible state. During this analysis, it became apparent that some instructions (e.g., halt) would not breach the fully-enclosed polyhedron, but would still produce a potential information flow. To handle this situation, a surface color was added to the fully-enclosed polyhedron to represent the externally visible state.

[77] This analysis is incomplete since it did not include the PPE processor.

# 6 EVALUATING FREESCALE P4080 USING THE FRAMEWORK

This chapter describes the analysis of the Freescale P4080 using the framework as originally defined and enhanced by the reanalysis of the CBEA described in the previous chapter.

## 6.1 IDENTIFY HARDWARE COMPONENTS

The first step in the framework is to identify the hardware components. The P4080 multicore architecture is designed for communication systems such as: routers, switches, base station controllers, and radio network controllers [ine]. Communication systems can be modeled using a control plane and a data plane. The control plane is intended to handle slow, complex packet management tasks, while the data plane is intended to handle common packet tasks at wire speed. Figure 17 illustrates an example architecture where the processor cores are logically partitioned into a control plane, a data plane, and other services.

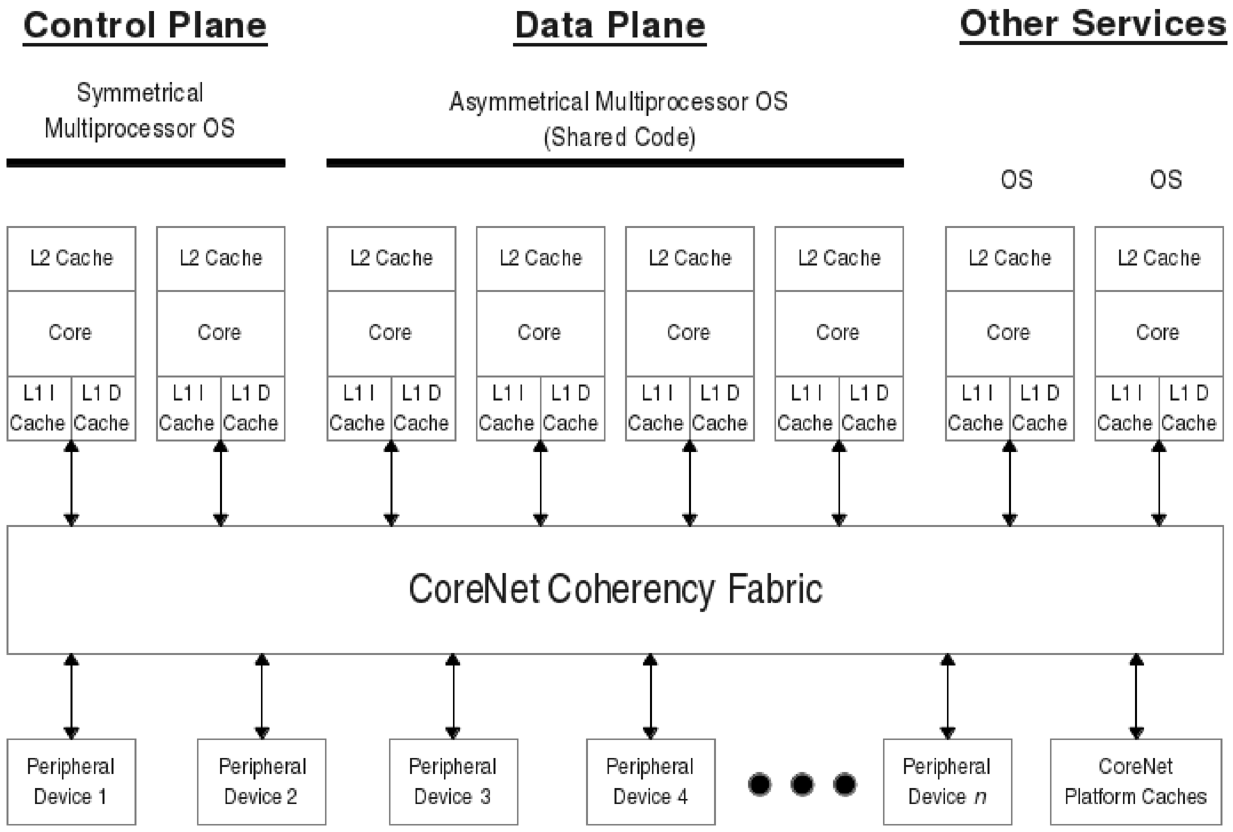


**Figure 17: P4080 Architecture**

The two cores in the control plane work together to form an independent Symmetric Multi-Processor (SMP) system. The four processor cores in the data plane run as independent systems.

The other two processor cores also run as independent systems and are responsible for all non-packet processing.

This configuration supports seven independent Asymmetric Multi-Processor (AMP) systems. Unlike SMP systems, AMP systems do not share a common resource manager, which greatly increases the difficulty of managing system resources and resolving resource conflicts [Fre11a]. To assist AMP configurations, the P4080 multicore architecture provides hardware support for partitioning systems resources (e.g., processor cores, system memory ranges, and peripherals) into logical partitions. System resources assigned to a logical partition are prevented by the hardware from accessing resources assigned to a different logical partition.

### 6.1.1 Processor Cores

Each processor core provides six pipelined, superscalar execution units, private instruction and data L1 caches, a private unified L2 cache, private Memory Management Units (MMUs), and a CoreNet Bus Interface Unit.

Execution Units:
A superscalar processor is capable of issuing multiple independent instructions into independent execution units in a single clock cycle. Each e500mc processor core implements the following execution units: two simple integer instruction units (SFX0, SFX1), a complex integer instruction unit (CFX), a branch unit (BU), a floating-point unit (FPU), and a load/store unit (LSU). The functionality of these six execution units are described below:

1. Simple Integer Units (SFX0, SFX1) - The SFX0 and SFX1 execution units process all the simple integer instructions. The simple integer instructions include all integer instructions except the integer multiplication and integer division instructions. The SFX0 execution unit is capable of processing the entire set of the simple integer instructions; whereas the SFX1 execution unit is only capable of executing a subset of the simple integer instructions.
2. Complex Integer Unit (CXF) - The CXF execution unit processes the integer multiplication and integer division instructions.
3. Branch Unit (BU) - The BU execution unit processes the branch and logical control register instructions.
4. Floating Point Unit (FPU) - The FPU execution unit processes the floating point instructions.
5. Load/Store Unit (LSU) - The LSU execution unit processes integer and floating-point load operations. The LSU manages transactions between the caches and the general purpose and floating point registers. The LSU coordinates traffic in the instruction pipeline with the load and store memory traffic to ensure that the processor core maintains a coherent and consistent view. The LSU is also responsible for calculating effective addresses, handling data alignment, and interfaces with the CoreNet bus interface unit [Fre11a].

To improve performance, all six execution units are pipelined. Pipelining breaks the instruction execution into discrete stages. These multiple stages simplify the execution sequence, allowing for a higher clock frequency. Instructions must pass sequentially through each stage to complete the instruction execution. Although each instruction requires multiple clock cycles to complete, a full pipeline completes instructions every clock cycle.

Caches:

Each e00mc processor core supports a two-level cache hierarchy. The first level (L1) cache is split into private data and instruction caches. The second level (L2) backside cache is also private to each e500mc processor core. This cache is physically implemented as a dynamic Harvard architecture (i.e., a single, unified cache for both instructions and data, but provides Harvard architecture properties by providing different behaviors for the instruction and the data cache entries). Instructions stored into the L2 cache are marked with the N status bit to denote that the cache line was loaded incoherently.

Memory Management Units:

The e500mc supports four different address types: Real Address (RA), Logical Address (LA), Effective Address (EA), and Virtual Address (VA). The processor core uses the RA type to interact with system memory (i.e., the CoreNet address space). LA types are used by guest operating systems. The hypervisor translates the LA to the RA when the guest writes the TLB entry. EA types are used by software to reference storage locations (e.g., instructions and data addresses). VA types are formed using EA types and supplying state information from the current address space context [Fre11b]. Equation (1) shows how the e500mc constructs normal VA entries by concatenating the following information together:

$$VA = MSR[GS] \,||\, LPIDR \,||\, MSR[IS|DS] \,||\, PID \,||\, EA \qquad \text{Eqn. (1)}$$

The values in Eqn. (1) are defined as:

1. MSR[GS] identifies the processor state (i.e., guest or hypervisor).
2. LPIDR identifies the logical partition in execution.
3. MSR[IS|DS] identifies if the VA is part of instruction or data address space.
4. PID values further identify the process address space.
5. EA is the effective address space.

The e500mc also supports VA using an external PID (Process Identification Number) addressing. External PID addressing provides a more efficient method for system software to move data and to perform cache operations across disjoint address spaces. Eqn. (2) shows the e500mc constructs VA entries using external PID addressing:

$$VA = EGS \,||\, ELPIDR \,||\, EAS \,||\, EPID \,||\, EA \qquad \text{Eqn. (2)}$$

The values in Eqn. (2) are defined as:

- EGS identifies the external guest state.
- ELPIDR identifies the external logical partition ID.
- EAS identifies the external context address space.
- EPID identifies the external context process ID.

- EA identifies the effective address space.

To maximize performance, each processor core implements a two-level, hierarchical MMU architecture. Two first-level MMUs are implemented in hardware. One MMU is dedicated for instructions, while the other MMU is dedicated for data. The L1 MMUs are maintained by hardware and are invisible to the software programming model. Hardware maintains L1 MMU entries as a proper subset of the second-level L2 MMU [Fre11a].

Translation Lookaside Buffers (TLB) are the hardware resources used to store the address translations, verify access controls, and maintain memory and cache attributes for the virtual memory pages. To maximize performance and flexibility, a variable-sized page and fixed-sized page TLB are implemented for each cache in the e500mc processor core. Both the variable-sized page TLB and the fixed-sized page TLB are searched in parallel for TLB hits. The L1 instruction MMU and L1 data MMU operate independently and can be accessed in parallel [Free11a]. The independent L1 instruction and L1 data TLBs presented in the e500mc processor core violate the TLB concept presented in [Fre11b]. The TLB concept states: "TLBs are defined by the architecture to be unified between instruction and data accesses. That is, a separate set of TLB structures between instruction fetch and data accesses are not permitted." Although not explicitly stated we believe this was seen as a necessary improvement for multicore architectures. This type of change leads to potential confusion and problems among security analysts who must be sure the documentation being used matches the microprocessor variant being analyzed.

<u>CoreNet Bus Interface Unit:</u>
The Bus Interface Unit (BIU) interfaces the processor cores to the CoreNet coherency fabric. The BIU handles CoreNet transaction ordering and maintains system coherency by address snooping and updating the affected processor components (e.g., caches, memory subsystems, Etc.).

### 6.1.2 CoreNet

<u>CoreNet Coherency Fabric:</u>
The CoreNet coherency fabric serves as a central interconnect for processor cores, platform-level caches, memory subsystems, peripheral devices, and I/O host bridges [Free11c]. The main purpose of the CoreNet coherency fabric is to provide the communication channel to move data from the source component to the destination component.

<u>CoreNet Coherency Cache:</u>
The frontside L3 cache (a.k.a., CoreNet platform cache) connects the memory controllers to the CoreNet coherency fabric. The CoreNet platform cache can be configured in one or more of the following modes: (1) a general purpose write-back cache, (2) an I/O stash, or (3) a memory mapped SRAM [Fre11c]. When the CoreNet platform cache has a backing store (i.e., configured in the general purpose writeback cache mode or the I/O stash mode), the CoreNet platform cache is then only able to cache address ranges present in the memory controller behind it. Each mode defines the allowable information flows and the L3 cache color.

<u>Peripheral Access Management Unit:</u>

Each processor core's MMU settings determine which memory regions are accessible to each core. Peripheral Access Management Units (PAMU) isolate CoreNet from other coherency and I/O domains. In order to prevent non-processor core system masters from accessing sensitive memory regions, the PAMUs enforce authentication and access controls to various memory regions.

Each non-processor core master is assigned a Logical I/O Device Number (LIODN). Non-processor device masters are assigned a unique LIODN by the hypervisor, providing an authentication method. Memory address requests are authorized against PAMU configured memory regions.

### 6.1.3 DDR2/DDR3 SDRAM Controller

The P4080 multicore architecture supports two fully programmable Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM) controllers. These DDR SDRAM controllers support Error Checking and Correction (ECC) memory to reduce bit error rates. The DDR SDRAM controllers support special features, including ECC error injection, to support system debugging.

### 6.1.4 Enhanced Local Bus Controller

The enhanced Local Bus Controller (eLBC) provides access to miscellaneous peripherals from the CoreNet Coherency Fabric. These miscellaneous devices include, but are not limited to, boot flash, security monitor, Universal Asynchronous Receiver/Transmitter (UART), General Purpose Input/Output (GPIO) pins, Serial Peripheral Interface (SPI) bus, Universal Serial Bus (USB) and Secure Digital / MultiMedia Cards (SD/MMC).

The eLBC is composed of eight local memory banks, a memory controller, and five peripheral interface controllers. The memory controller is responsible for managing the eight memory banks shared by the peripheral interface controllers. The peripheral interface controllers are composed of one General Purpose Chip Select Machine (GPCM), one NAND Flash Control Machine (FCM), and three User-Programmable Machines (UPMs). Each memory bank is assigned to at most one of the peripheral interface controllers. The address and data lines are shared by all the peripheral interface controllers. The memory bank containing the address determines which peripheral interface controller masters the local bus.

### 6.1.5 High-Speed Peripheral Interface Complex

<u>Peripheral Component Interconnect Express:</u>
The Peripheral Component Interconnect Express (PCIe) bus provides a high-performance, point-to-point topology. Each end point device is connected to the root complex (host) via a serial interface. PCIe provides a standard interface and has a significant install base.

The P4080 architecture provides three PCIe controllers. Each PCIe controller can be configured to operate either as a root complex or an end point device.

<u>Rapid Input/Output:</u>

The Rapid Input/Output (RapidIO) interconnect architecture provides a high-performance, packet-switched, interconnect technology. RapidIO offers a standard interface and has a significant install base in embedded production systems. The P4080 architecture provides two serial RapidIO (sRIO) endpoints and the RapidIO Message Unit (RMU). These RapidIO units are compliant with the RapidIO Interconnect Specification, Revision 1.2.

Direct Memory Access Controllers:
The P4080 architecture provides two Direct Memory Access (DMA) controllers. Each DMA controller provides four DMA channels. Each DMA channel is capable of complex data moving and advanced transaction chaining.

A DMA transfer request can be initiated by either a processor core or an external host. Either a processor core or a new external device can initiate a DMA transfer. Once the DMA transaction has been accepted, a DMA channel is assigned and remains active until either the DMA transaction is completed or the allotted DMA channel bandwidth is consumed.

### 6.1.6 Data Path Acceleration Architecture:

The Data Path Acceleration Architecture (DPAA) provides the infrastructure for the P4080 multicore architecture to support simplified sharing of network interfaces and hardware accelerators by multiple processor cores. Traditional routing and bridging, firewalls, Virtual Private Network (VPN) terminations, intrusion detection and prevention systems, and network anti-virus solutions illustrate some example applications that the DPAA was designed to accelerate. The DPAA typically manages the packet distribution and queue congestion, leaving the software in charge of protocol processing [Fre11d].

The DPAA uses components to provide a scalable, modular solution for different processor families. These components are identified as either core or hardware accelerator components. The core components include the Buffer Manager (BM), the Queue Manager (QM), and the Frame Manager (FM). The P4080 multicore architecture provides a single BM, a single QM, and two FMs. Additionally, the P4080 multicore architecture also provides two hardware accelerator components: the Security Encryption Engine (SEE) and the Pattern Matching Engine (PME). Figure 18 illustrates the subsection of the P4080 multicore architecture representing these DPAA components. Table 21 provides common terms and definitions used in the DPAA.

**Table 21: DPAA Terms and Definitions**

| Term | Definition |
|---|---|
| Buffer | Region of contiguous memory managed by the DPAA BM |
| Buffer Pool | Set of Buffers with common characteristics (e.g., size, alignment, access controls) |
| Frame | Single Buffer or list of Buffers that hold data, (e.g., packet payload, header or other control information) |
| Frame Queue | FIFO of Frames |
| Work Queue | FIFO of Frame Queues |
| Channel | Set of eight Work Queues with hardware provided priority access |

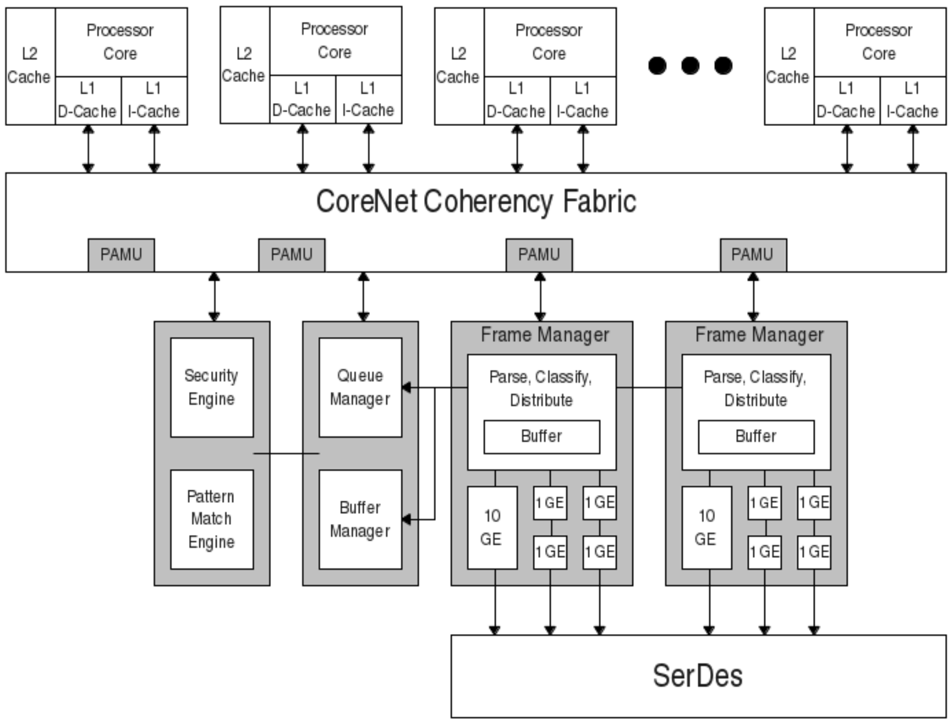


**Figure 18: DPAA Block Diagram**

<u>Buffer Manager:</u>

The primary function of the BM is to reduce the software overhead for managing the free buffer pools for multiple processor cores, network interfaces, and other hardware accelerators. The BM stores free buffer proxy records in a private external memory region. The BM is programmed to use this private external memory region via the FBPR_BARE, FBPR_BAR, and FBPR_AR registers. Additionally, a CoreNet Local Access Window (LAW) must be configured for this private external memory region. This CoreNet LAW number must also be programmed into the BMAN_LIODNR register. This configuration is used by the QM to generate authenticated CoreNet transactions to the private external memory region for the BM [Fre11d]. Figure 19 provides a block diagram of the BM component.

Software allocates memory buffers and passes these memory buffers to the BM, which maintains them as free lists. The BM supports 64 buffer pools, allowing the software to organize memory buffers by different characteristics (e.g., buffer size, memory partition, address alignment). The BM does not enforce these buffer characteristics; this can be made the software's responsibility if desired.

To reduce buffer request latency, the BM maintains a small stockpile of buffers for each buffer pool. The BM maintains this stockpile by retrieving additional buffers from the external memory buffer list when the stockpile falls below a software configured limit. The BM will also push buffers back to the external memory buffer lists when the number of internal buffers exceeds a software configured limit.

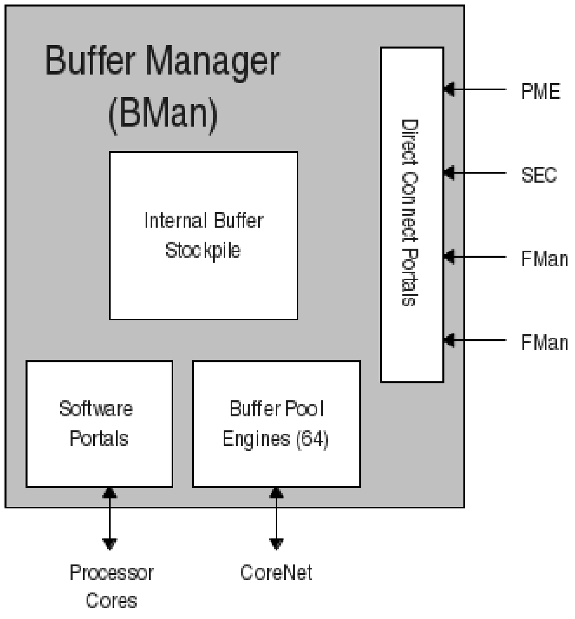


**Figure 19: DPAA BM Block Diagram**

Direct connect portals bypass the system interconnect bus (i.e., CoreNet) to reduce latency for hardware components participating in the DPAA. The P4080 multicore architecture provides direct connect portals to each frame manager, the security engine, and the pattern matching engine. Direct connect portals use a dedicated set of interface signals to handle buffer states. This allows the hardware component to implement automatic flow-control or other actions depending upon the buffer state [Fre11d].

The BM provides 10 software portals for the processor cores. Software can perform the following functions on each software portal: (1) configure the software portal (including how interrupts are handled), (2) query the buffer pools to see which ones have free buffers available, (3) acquire buffers from the pool, and (4) return buffers to the pool.

Queue Manager:
The primary function of the QM is to provide a central resource for efficiently moving data between multiple processor cores, network interfaces, and other hardware accelerators. To manage this data, the QM uses frames, frame queues, work queues, channels, portals, algorithmic sequencers, and a multi-way resource arbiter. A frame is the basic unit of data managed by the QM. Each frame contains one or more buffers. Frame queues provide the basic queuing structure used in the QM. Frame queues are created by software and managed by the QM using frame queue descriptors. Frame queues provide three distinct mechanisms for congestion management WRED, congestion state tail drop, and frame queue tail drop. When a frame queue is ready to be processed, it is enqueued into a work queue. Work queues are organized into channels. Each channel has eight work queues with a relative priority of zero to seven. The QM, in the P4080 multicore architecture, supports 51 channels: 12 dedicated channels for each FM, one dedicated channel for the SEC, one dedicated channel for the PME, 10 dedicated channels for the software portals, and 15 pool channels shared by all the software portals. Portals provide the external interface to QM. QM provides 10 software portals and four

direct connect portals. The P4080 multicore architecture also contains 16 algorithmic sequencers. These algorithmic sequencers manage the interaction between the QM channels and portals. Algorithmic sequencers provide both isolation and parallelism between the portals. The multi-way resource arbiter serves as the central entity between the algorithmic sequencers and the portals and the QM resource managers. This allows the algorithmic sequencers to act independently while maintaining the integrity of the QM resources. Figure 20 provides a block diagram of the QM component.

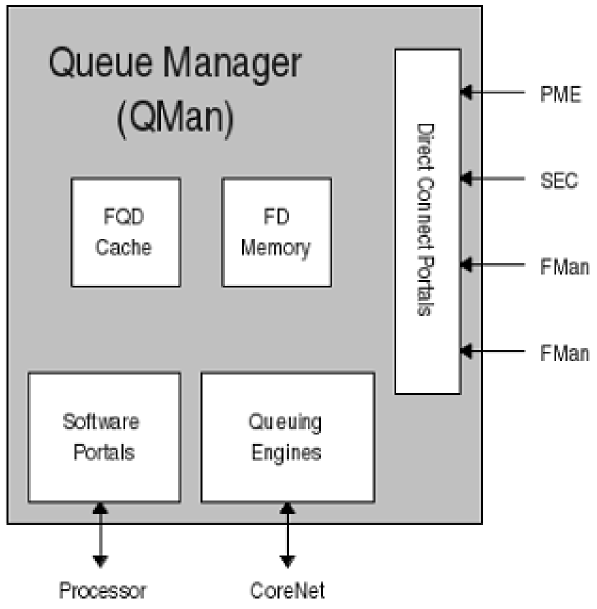


**Figure 20: DPAA QM Block Diagram**

The QM requires two external memory regions. One memory region is reserved for frame queue descriptors. The other memory region is reserved for the QM to store its private member data structures. Software programs the FQD_BAR, FQD_BARE, BQD_AR, PFQD_BAR, PFQD_BARE, and PQFD_AR registers to configure the QM to use these two memory regions. The QM's private memory region must be protected from all access other than the QM.

Frame Manager:

The P4080 multicore architecture provides two FM components. The primary function of an FM is to combine Ethernet network interfaces with packet distribution logic in order to produce intelligent distribution and queuing decisions for incoming traffic at line rate. Figure 21 provides a block diagram for the FM component.

Each FM component provides four variable speed Ethernet controllers (1GE) and one 10-Gigabit Ethernet Controller (10GE). The variable speed Ethernet controllers are configurable to run at 10 Mbps, 100 Mbps, and 1 Gbps. These Ethernet controllers are connected to the SerDes bus and are responsible for the transmission and reception of network packets. The remaining blocks in Figure 21 are used to accelerate the parse, classify, and distribute flows. The BMI block interfaces with the DPAA BM component to acquire and release buffers. The Queue Manager Interface (QMI) block interfaces with the DPAA QM component to enqueue and dequeue frames. The classifier block provides a coarse classification by extracting common protocol fields from the packet. The DMA block moves data between external memory and the internal FM shared memory buffer. The Internal Shared Memory Buffer block stores the transmit and receive FIFOs and the coarse classification data structures. The scheduler manages the frame processing between the internal FM blocks. The Parser block provides finer-grained parsing capabilities. This block supports both hardwired and user-defined parsing functions. The KeyGen block generates frame queue identifiers based on user-defined criteria. The Policer block supports the differentiation of services at wire speed. This block may also be used to protect the processor cores from excessive traffic or packet rates.

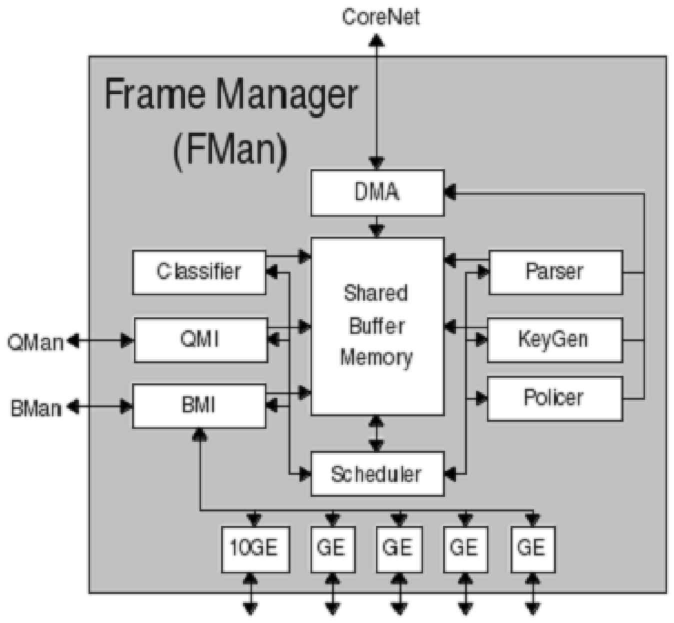


**Figure 21: DPAA FM Block Diagram**

Pattern Matching Engine:
The primary purpose of the PME is to search DPAA data streams against patterns constructed from regular expressions at wire speed. Figure 22 provides a block diagram of the PME component.

The Pattern Matcher Frame Agent (PMFA) dequeues any available frames from the QM and determines what action needs to be taken on the frame(s). When a scan request is received, the frame is separated into work units (i.e., atomic PME work request operations). The PMEA preserves order, so work units are completed in the same order they were selected. The PMFA schedules the work units into the pipeline for additional processing.

The PME implements the core pattern matching functionality as a three stage pipeline. The first stage is implemented in the Key Element Scanner (KES). The KES searches up to 32,000 patterns simultaneously using a proprietary multistage hash algorithm. Work units generating a hash match are forwarded to the next stage where a more stringent comparison is performed.

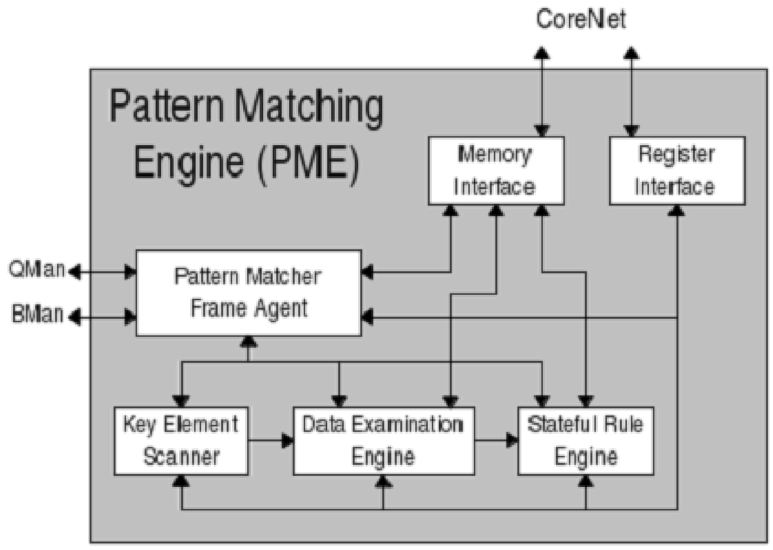


**Figure 22: DPAA PME Block Diagram**

The second stage in the pipeline is the Data Examination Engine (DXE). This stage implements a non-deterministic finite automation capable of implementing a significant subset of the regular expression pattern definition syntax. Successfully matched work units are passed to the final stage in the pipeline.

The Stateful Rule Engine (SRE) is the final stage in the pipeline. This stage implements a finite state machine to track complex scenarios involving one or more flows. For example, the SRE could be used to detect patterns in a pair of flows. Each flow would represent the communication direction. The SRE reports pattern match events to the PMFA, which in turn notifies software of the scan results.

The memory interface provides the PME components access to initiate read and write CoreNet transactions. Software must configure a valid Logical I/O Device Number (LIODN) in the PME LIODNR register.

The Register Interface provides software access to all the PME registers.

Security Encryption Engine:
The primary purpose of the Security Encryption Engine (SEE) is to provide hardware acceleration to cryptographic algorithms. The SEE implements block encryption algorithms, stream cipher algorithms, hashing algorithms, public key algorithms, runtime integrity checking, and a hardware random number generator [Fre12]. Figure 23 provides a block diagram of the SEE component.

The descriptor is a SEE control structure that instructs the cryptographic hardware modules in performing a single task. Each descriptor consists of a sequence of functional commands. Conditional and unconditional jumps are permitted to change the program flow or to jump to a different descriptor. Three types of descriptors are available: job descriptor, trusted descriptor, and shared descriptor. Job descriptors are typically created by software or via the DPAA. Job descriptors do not have access to the trusted descriptor-only black keys, trusted descriptor-only blobs, or restricted memory and devices addresses. Job descriptors are also not integrity checked at runtime. Trusted descriptors must be created in a specially-privileged job ring, are runtime integrity checked, and do have access to the trusted descriptor-only black keys, trusted descriptor-only blobs, and restricted memory and device addresses. Shared descriptors are constructed to be fetched once and then used by many tasks [Fre12].

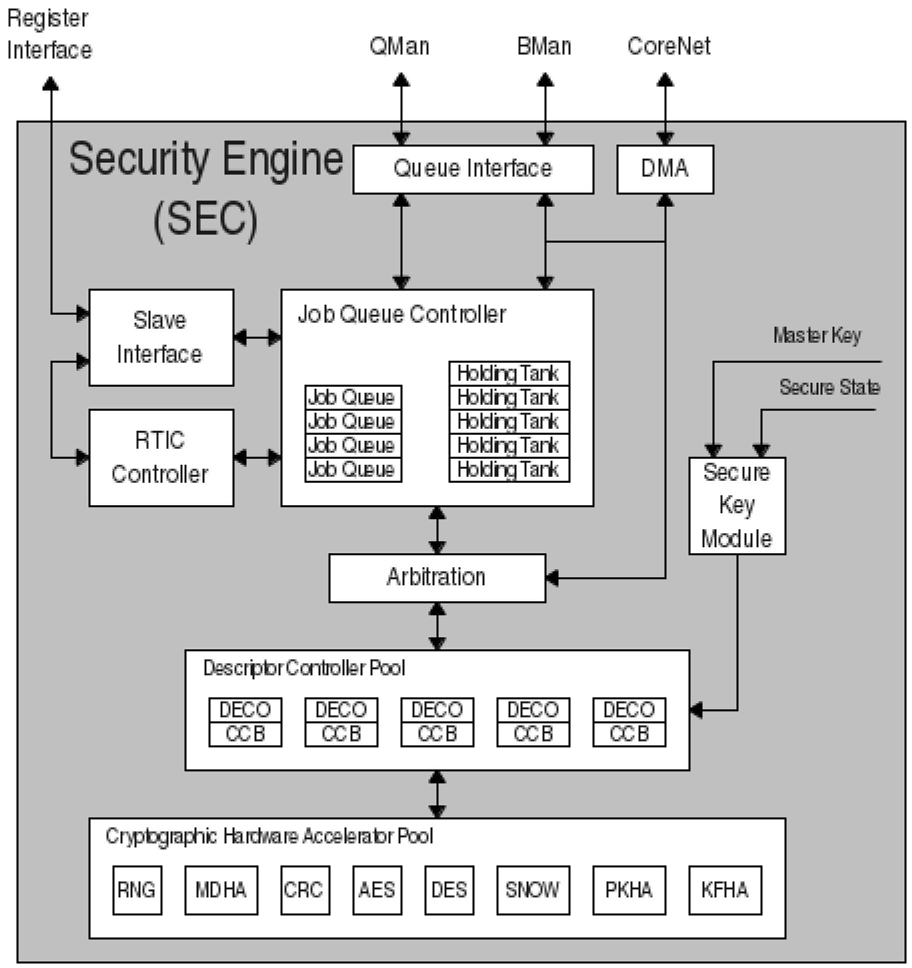


**Figure 23: DPAA SE Block Diagram**

The job queue controller serves as the descriptor scheduler for the SEE. The job queue controller processes jobs using a round-robin scheduler, pulling jobs from the job ring, from the DPAA Queue Manager Interface, and then from the Real Time Integrity Checker (RTIC). The job queue controller pre-fetches some or all of the descriptors and places them into the holding tanks. This pre-fetching permits the job queue controller to take advantage of shared descriptors when allocating jobs to descriptor controller pool [Fre12].

Descriptor controllers (DECO) are responsible for the execution of job descriptors. During job descriptor execution, the DECO invokes the DMA controller to read and write to system memory addresses specified in the job descriptor. The DECO also needs access to the Cryptographic Hardware Accelerators (CHA). The CHA Cluster Block (CCB) contains all the hardware required to control the CHA units. Arbitration for shared CHAs is automatically handled [Fre12]. Table 22 lists the hardware accelerator and the supported cryptographic algorithms.

**Table 22: Cryptographic Hardware Accelerators**

| Abbrev. | Definition | Algorithms |
|---|---|---|
| PKHA | Public-Key hardware accelerator | RSA, Diffie-Hellman, DSA, Elliptic-Curve Diffie-Hellman, Elliptic-Curve DSA |
| KFHA | Kasumi f8 & f9 hardware accelerator | Kasumi f8 encryption, Kasumi f9 authentication |
| AFHA | ARC-four hardware accelerator | Alleged RC4 encryption |
| DESA | Data encryption standard accelerator | DES and Triple-DES encryption |
| RNG | Random number generator | True hardware random, pseudo-random number generators |
| SNOWf8 | SNOW 3G f8 accelerator | SNOW f8 encryption |
| SNOWf9 | SNOW 3G f9 accelerator | SNOW f9 authentication |
| MDHA | Message-digest hardware accelerator | MD-5, SHA-1, SHA-256, SHA-384 and SHA-512 |
| AESA | AES accelerator | AES encryption |

The RTIC participates in the trust architecture to assist with boot authentication and to verify the integrity of the system memory. The RTIC is capable of verifying memory contents at both system boot and runtime. The RTIC cooperates with a high-assurance, secure boot loader. The secure boot loader validates the initial code to execute, then enables the RTIC to hash the normal boot image. Once the boot image hash has been calculated, the RTIC generates an interrupt, allowing the secure boot loader to compare the calculated hash against a digitally signed hash value stored in the secure boot loader. The secure boot loader decides what action to take in the event of a hash mismatch. After the secure boot loader has verified the boot image, software can put the RTIC into runtime mode to periodically verify the integrity of specific blocks of system memory. If the RTIC detects a runtime hash mismatch, an interrupt is generated and a security violation is signaled to the security monitor hardware [Fre12].

The secure key module also participates in the trust architecture. On power-on reset, the SEE generates new random keys for the Job Descriptor Key Encryption Key (JDKEK), the Trusted Descriptor Key Encryption Key (TDKEK), and the Trusted Descriptor Signing Key (TDSK). Both trusted and job descriptors are permitted to encrypt and/or decrypt normal black keys using JDKEK, but only trusted descriptors are permitted to encrypt and/or decrypt trusted black keys using TDKEK. The TDSK is used to verify trusted descriptors by computing a keyed hash over the trusted descriptor [Fre12].

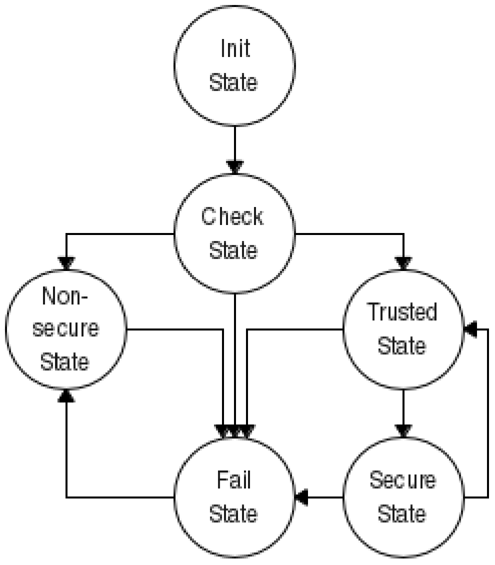


**Figure 24: Security Mode State Machine**

These secure keys are influenced by the security monitoring hardware state. The SEESEC operates in one of the following security modes: trusted mode, secure mode, non-secure mode, and fail mode. Figure 24 provides a state diagram showing security monitor states and transitions.

The SEE starts in the *Init State* and transitions to the *Check State* upon power-on reset. Both of these states have the SEE operating in the non-secure mode. The *Check State* performs various hardware and software security checks. If any of these checks fail, the *Check State* transitions to the *Non-secure state* and remains in the non-secure mode. If all the various hardware and software security checks pass, then the *Check State* transitions to the *Trusted State*. At this point, the SEE is operating in the trusted mode. Software can choose to transition between the *Trusted State* and the *Secure State*. The SEE operates in secure mode when in the *Secure State*. If the SEE fails specific hardware security checks, the SEE transitions into the *Fail State* and operates in the fail mode. In this mode, all the critical security parameters (including JDKEK, TDKEK, and TDSK) are zeroed in accordance with FIPS-140-2. Software may transition from the *Fail State* to the *Non-secure state*, but the critical security parameters will not be restored until the next power-on reset [Fre12].

### 6.1.7 Real Time Debug:

Not all of the documentation for the Real Time Debug block is publicly available at the time of this report writing. The *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual* identifies the *Advanced QorIQ Debug and Performance Monitoring Guide* for providing the complete configuration and use of chip-level debug and performance monitoring functionality. Specifically, Chapter 3 is listed to be documentation describing the Debug Control and Status Registers (DCSR). These DSCR exist in a separate memory map and are devoted to debug functionality in the P4080 multicore architecture. Some debug functionality is mapped

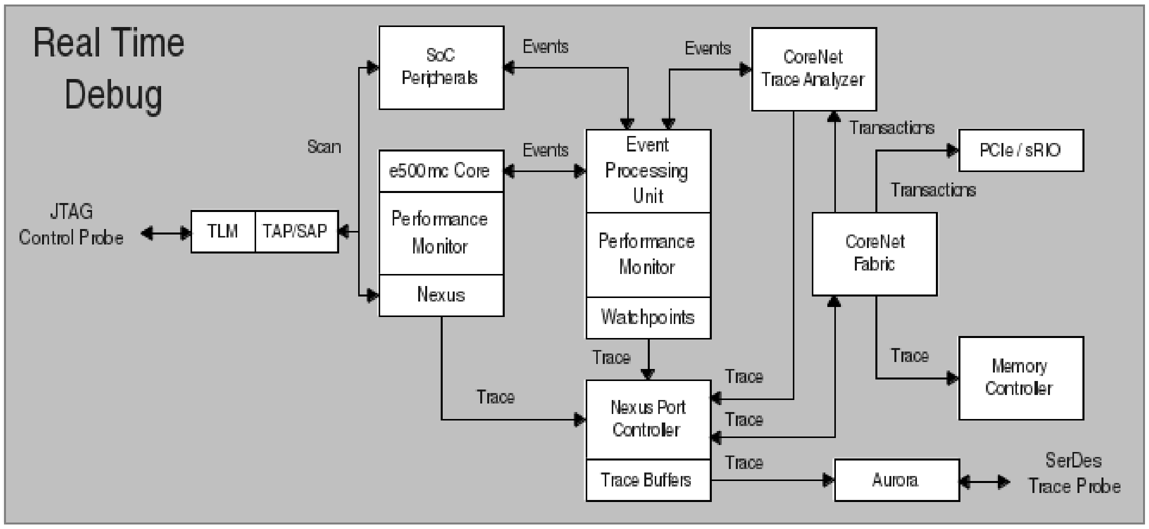


**Figure 25: Simplified Real Time Debug Block**

into the Configuration, Control, and Status Register (CCSR) memory space [Fre11c]. Figure 25 shows a simplified diagram of the major debug components.

The P4080 multicore architecture is populated with debug and performance monitoring instrumentation. Debug instrumentation is divided into two categories: function-specific and cross-functional. Function-specific debug logic is specialized to help gather data and control a specific functional unit. The P4080 multicore architecture supplies function-specific debug logic for the following components: CoreNet coherency fabric, CoreNet platform caches, DDR external memory, FM, QM, and SEE DPAA components, PCI Express and serial Rapid IO, and e500mc processor cores. Each of these functional areas will then forward debug information to the cross-functional facilities. The cross-functional components coordinate run control, performance monitoring, and tracing operations for the entire P4080 multicore architecture. Cross-functional components include the Event Processing Unit (EPU), Nexus Port Controller (NPC), Nexus Concentrator (NXC), and Nexus Aurora Link (NAL).

### 6.1.8 Hardware Component List

Table 23 provides a list of the independent, system-high components evaluated for this chapter.

## 6.2 IDENTIFY AND EVALUATE INFORMATION FLOWS

The second step of the framework is to identify information flows and safeguards.

### 6.2.1 Processor Core

The primary function of a processor core is to perform work by executing processor instructions. The processor core does this by continually fetching and executing processor instructions. The Program Counter (PC) contains the memory address of the next instruction to fetch. Typically, after each instruction fetch, the program counter is incremented to fetch the next sequential memory instruction. The program counter may be altered by program flow control logic (e.g., branch instructions) or interrupts. Interrupts cause the processor core to save context state (MSR register) and the next instruction address. The program counter is changed to a predetermined interrupt handler [Fre11a].

**Table 23: P4080: Hardware Component List**

| Hardware Component | Evaluated |
|---|---|
| Processor Cores (8) | Yes |
| CoreNet<br>Cornet Coherency Fabric<br>CoreNet Platform Caches (2) | Yes |
| DDR2/DDR3 SDRAM Controllers (2) | Yes |
| Enhanced Local Bus Controller<br>Peripheral Controllers | Yes<br>No |
| High Speed Peripheral Interface Complex<br>PCI Express Controllers (3)<br>RapidIO Message Unit<br>Serial RapidIO Endpoints (2)<br>Direct Memory Access Controllers (2) | Yes |
| Data Path Acceleration Architecture<br>Buffer Manager Queue Manager Frame Managers (2)<br>Pattern Matching Engine<br>Security Encryption Engine | Yes |
| Real Time Debug | Yes |

In addition to executing processor instructions, each processor core also provides three sets of registers. The general purpose and floating-point registers are used by the integer and floating-point execution units. Values in the general purpose and floating-point registers do not affect the externally visible state of the processor. Values in the special-purpose registers may cause externally visible state changes.

#### 6.2.1.1 Information Flows

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

- Any information flow breaching any surface of the polyhedron must be evaluated.
- Any information flow altering the externally visible state must be evaluated.
- Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

Instructions:
The e500mc processor instruction set analysis provides 227 instructions that are categorized into 14 categories. Each instruction category was analyzed looking for potential information flows or externally visible state changes. Since the hypervisor is assumed to be trusted, instructions requiring hypervisor privilege are ignored in this analysis. Instruction categories that cause potential information flows or that alter externally visible states are presented in this chapter. The other instruction categories are included Appendix B for completeness.

The Branch and Flow instructions (Appendix B, Table 60) affect the state of the program counter, count register, and link register. These instructions do not generate information flows breaching the polyhedron surface nor do they alter the externally visible state.

The Conditional Branch Control instructions (Table 24) require additional scrutiny. The **twi** and **tw** instructions cause the program to generate an exception to either IVOR6 (Program) or IVOR15 (Debug). While these instructions do not generate information flows breaching the polyhedron surface, the hypervisor should pay special attention to how the IVOR6 and IVOR15 are setup to ensure that a timing covert communication channel is not present.

**Table 24: Conditional Branch Control Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Conditional Register AND | crand | User |
| Conditional Register OR | cror | User |
| Conditional Register XOR | crxor | User |
| Conditional Register NAND | crnand | User |
| Conditional Register NOR | crnor | User |
| Conditional Register Equivalent | creqv | User |
| Conditional Register AND with Complement | crandc | User |
| Conditional Register OR with Complement | crorc | User |
| Move Condition register Field | mcrf | User |
| Trap Word Immediate | twi | User |
| Trap Word | tw | User |
| Integer Select | isel | User |

The Debug instructions (Table 25) require additional scrutiny. When an external debugger is attached and the e500mc core is configured to halt on the execution of the **dhn** instruction, this instruction causes the e500mc processor core to halt and wait for the external debugger. If the **dhn** instruction is executed when an external debugger is not attached or if the e500mc core is not configured to halt on the execution of the **dhn** instruction, an illegal instruction exception is generated. While this instruction does not generate an information flow breaching the polyhedron surface, the hypervisor should pay special attention to ensure this instruction cannot be used as a covert communication channel.

**Table 25: Debug Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Debug Notify Halt | dnh | User |

The Memory Synchronization instructions (Table 26) require additional scrutiny. These instructions control memory operation completion and visibility to the other components accessing memory. The **mbar** and **msync** instructions support options to broadcast memory coherency information on the CoreNet interface.

**Table 26: Memory Synchronization Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Instruction Synchronization | isync | User |
| Load Word and Reserve Indexed | lwarx | User |
| Memory Barrier | mbar | User |
| Memory Synchronize | msync | User |
| Store Word Conditional Indexed | stwcx. | User |

The Floating-Point instructions (Appendix B, Table 61), Floating-Point Arithmetic instructions in Table 63, Floating-Point Compare instructions in Table 64, Floating-Point Rounding and Conversion instructions in Table 65, and Floating-Point Status and Control Register instructions (Appendix B, Table 62) do not generate information flows breaching the polyhedron surface nor do they alter the externally visible state.

The Processor Control instructions (Table 27) require additional scrutiny. The **mtspr** writes values to the special purpose registers. The special purpose registers are described in additional detail after the processor instruction analysis is complete. Modification of the special purpose register values may cause an information flow to breach the polyhedron surface or alter the externally visible state.

**Table 27: Processor Control Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Move to Conditional Register Fields | mtcrf | User |
| Move to Conditional Register from XER | mcrxr | User |
| Move from Condition Register | mfcr | User |
| Move from Once Condition Register Field | mfocrf | User |
| Move to One Condition Register Field | mtocrf | User |
| Move to Special-Purpose Register | mtspr | User |
| Move from Special-Purpose Register | mfspr | User |
| Move from Time Base | mftb | User |
| Wait for Interrupt | wait | User |
| Move from Performance Monitor Register | mfpmr | User |
| Move to Performance Monitor Register | mtpmr | Guest Supervisor |

The Integer instructions (Appendix B, Table 66) do not generate information flows that breach the polyhedron surface nor do they alter the externally visible state.

The Load and Store instructions (Table 28) generate information flows breaching the polyhedron surface. The purpose of these instructions is to move information to and from the main memory hierarchy. The e500mc also supports decorated load and store instructions and external PID load and store instructions. Decorated load and store instructions provide efficient access to System-On-a-Chip (SoC) specific storage addresses. External PID instructions are used to load and store addresses into a different address space.

**Table 28: Load and Store Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Load Byte and Zero | lbz | User |
| Load Byte and Zero Indexed | lbzx | User |
| Load Byte and Zero with Update | lbzu | User |
| Load Byte and Zero with Update Indexed | lbzux | User |
| Load Half Word and Zero | lhz | User |
| Load Half Word and Zero Indexed | lhzx | User |
| Load Half Word and Zero with Update | lhzu | User |
| Load Half Word and Zero with Update Indexed | lhzux | User |
| Load Half Word Algebraic | lha | User |
| Load Half Word Algebraic Indexed | lhax | User |
| Load Half Word Algebraic with Update | lhau | User |
| Load Half Word Algebraic with Update Indexed | lhaux | User |
| Load Word and Zero | lwz | User |
| Load Word and Zero Indexed | lwzx | User |
| Load Word and Zero with Update | lwzu | User |
| Load Word and Zero with Update Indexed | lwzux | User |
| Store Byte | stb | User |
| Store Byte Indexed | stbx | User |
| Store Byte with Update | stbu | User |
| Store Byte with Update Indexed | stbux | User |
| Store Half Word | sth | User |
| Store Half Word Indexed | sthx | User |
| Store Half Word with Update | sthu | User |
| Store Half Word with Update Indexed | sthux | User |
| Store Word | stw | User |
| Store Word Indexed | stwx | User |
| Store Word with Update | stwu | User |
| Store Word with Update Indexed | stwux | User |
| Load Half Word Byte-Reversed Indexed | lhbrx | User |
| Load Word Byte-Reversed Indexed | lwbrx | User |
| Store Half Word Byte-Reversed Indexed | sthbrx | User |
| Store Word Byte-Reversed Indexed | stwbrx | User |
| Load Multiple Word | lmw | User |
| Store Multiple Word | stmw | User |
| Load Floating-Point Single | lfs | User |
| Load Floating-Point Single Indexed | lfsx | User |
| Load Floating-Point Single with Update | lfsu | User |
| Load Floating-Point Single with Update Indexed | lfsux | User |
| Load Floating-Point Double | lfd | User |
| Load Floating-Point Double Indexed | lfdx | User |
| Load Floating-Point Double with Update | lfdu | User |
| Load Floating-Point Double with Update Indexed | lfdux | User |
| Store Floating-Point Single | stfs | User |

| | | |
|---|---|---|
| Store Floating-Point Single Indexed | stfsx | User |
| Store Floating-Point Single with Update | stfsu | User |
| Store Floating-Point Single with Update Indexed | stfsux | User |
| Store Floating-Point Double | stfd | User |
| Store Floating-Point Double Indexed | stfdx | User |
| Store Floating-Point Double with Update | stfdu | User |
| Store Floating-Point Double with Update Indexed | stfdux | User |
| Store Floating-Point as Integer Word Indexed | stfiwx | User |
| Load Byte with Decoration Indexed | lbdx | User |
| Load Half Word with Decoration Indexed | lhdx | User |
| Load Word with Decoration Indexed | lwdx | User |
| Load Floating-Point Double Word w/ Decoration Indexed | lfddx | User |
| Store Byte with Decoration Indexed | stbdx | User |
| Store Half Word with Decoration Indexed | sthdx | User |
| Store Word with Decoration Indexed | stwdx | User |
| Store Floating-Point Double Word with Decoration | stfddx | User |
| Decorated Storage Notify | dsn | User |
| Load Byte by External PID Indexed | lbepx | Guest Supervisor |
| Load Floating-Point Double Word by External PID | lfdepx | Guest Supervisor |
| Load Half Word by External PID Indexed | lhepx | Guest Supervisor |
| Load Word by External PID Indexed | lwepx | Guest Supervisor |
| Store Byte by External PID Indexed | stbepx | Guest Supervisor |
| Store Floating-Point Double Word by External PID | stfdepx | Guest Supervisor |
| Store Half Word by External PID Indexed | sthepx | Guest Supervisor |
| Store Word by External PID Indexed | stwepx | Guest Supervisor |
| Load Byte by External PID Indexed | lbepx | Guest Supervisor |
| Load Floating-Point Double Word by External PID | lfdepx | Guest Supervisor |
| Load Half Word by External PID Indexed | lhepx | Guest Supervisor |
| Load Word by External PID Indexed | lwepx | Guest Supervisor |
| Store Byte by External PID Indexed | stbepx | Guest Supervisor |
| Store Floating-Point Double Word by External PID | stfdepx | Guest Supervisor |
| Store Half Word by External PID Indexed | sthepx | Guest Supervisor |
| Store Word by External PID Indexed | stwepx | Guest Supervisor |
| Load Byte by External PID Indexed | lbepx | Guest Supervisor |
| Load Floating-Point Double Word by External PID | lfdepx | Guest Supervisor |
| Load Half Word by External PID Indexed | lhepx | Guest Supervisor |
| Load Word by External PID Indexed | lwepx | Guest Supervisor |
| Store Byte by External PID Indexed | stbepx | Guest Supervisor |
| Store Floating-Point Double Word by External PID | stfdepx | Guest Supervisor |
| Store Half Word by External PID Indexed | sthepx | Guest Supervisor |
| Store Word by External PID Indexed | stwepx | Guest Supervisor |
| Data Cache Block Flush by External PID Indexed | dcbfep | Guest Supervisor |
| Data Cache Block Store by External PID Indexed | dcbstep | Guest Supervisor |
| Data Cache Block Touch by External PID Indexed | dcbtep | Guest Supervisor |
| Data Cache Block | dcbtstep | Guest Supervisor |

|  |  |  |
|---|---|---|
| Data Cache Block Zero by External PID Indexed | dcbzep | Guest Supervisor |
| Data Cache Block Zero Long by External PID Indexed | dcbzlep | Guest Supervisor |
| Instruction Cache Block Invalidate by External PID Indexed | icbiep | Guest Supervisor |

The Memory Control instructions (Table 29) generate information flows breaching the polyhedron surface. These instructions control the instruction caches, the data caches, and the TLB entries. The **dcblc**, **dcbtls**, **dcbtstls**, **icbtls,** and **icblc** instructions have a configurable privilege level. Hypervisor privilege is required to execute these instructions when the UCLEP bit is set in the Machine State Register (MSR) and the Machine State Register Protect (MSRP) registers. The **tlbix** instruction will cause a hypervisor exception to be thrown if the DGTMI bit is set in the Embedded Processor Control Register (EPCR).

**Table 29: Memory Control Instructions**

| **Instruction** | **Mnemonic** | **Privilege Level** |
|---|---|---|
| Data Cache Block Allocate | dcba | User |
| Data Cache Block Allocate by Line | dcbal | User |
| Data Cache Block Flush | dcbf | User |
| Data Cache Block Set to Zero | dcbz | User |
| Data Cache Block Set to Zero by Line | dcbzl | User |
| Data Cache Block Store | dcbst | User |
| Data Cache Block Touch | dcbt | User |
| Data Cache Block Touch for Store | dcbtst | User |
| Instruction Cache Block Invalidate | icbi | User |
| Instruction Cache Block Touch | icbt | User |
| Data Cache Block Lock Clear | dcblc | User |
| Data Cache Block Touch and Lock Set | dcbtls | User |
| Data Cache Block Touch for Store and Lock Set | dcbtstls | User |
| Instruction Cache Block Lock Clear | icblc | User |
| Instruction Cache Block Touch and Lock Set | icbtls | User |
| Data Cache Block Invalidate | dcbi | Guest Supervisor |
| TLB Invalidate Local | tlbilx | Guest Supervisor |
| TLB Invalidate Virtual Address Indexed | tlbivax | Hypervisor |
| TLB Read Entry | tlbre | Hypervisor |
| TLB Search Indexed | tlbsx | Hypervisor |
| TLB Synchronize | tlbsync | Hypervisor |
| TLB Write Entry | tlbwe | Hypervisor |
| Message Clear | msgclr | Hypervisor |
| Message Send | msgsnd | Hypervisor |

The System Linkage instructions (Table 30) do not cause information flows that breach the polyhedron surface, but they may alter the externally visible state. The **rfi**, **rfdi**, **rfmci**, **rfci**, and **rfgi** instructions are all used to exit an interrupt state and return to normal processing mode. The **ehpriv** instruction calls the hypervisor Interrupt Vector Offset Register (IVOR). The **mfmsr** and **mtmsr** provide access to alter some bits in the MSR. The MSR presents a significant amount of flexibility and external visible state potential. The **wrtee** and **wrteei** instructions allow the guest supervisor to enable and/or disable external interrupts.

Registers:

Analysis of the e500mc processor separates the registers into 15 categories. Since the hypervisor is assumed to be trusted, registers requiring hypervisor privilege are ignored in this analysis. Register categories that cause potential information flows or that alter the externally visible state are presented in this chapter (Table 32 - Table 43). The other register categories are referenced in Table 31, but included in Appendix B for completeness.

**Table 30: System Linkage Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| System Call | sc | User |
| Return from Interrupt | rfi | Guest Supervisor |
| Return from Debug Interrupt | rfdi | Hypervisor |
| Return from Machine Check Interrupt | rfmci | Hypervisor |
| Return from Critical Interrupt | rfci | Hypervisor |
| Return from Guest Interrupt | rfgi | Guest Supervisor |
| Hypervisor Privilege | ehpriv | Guest Supervisor |
| Move from Machine State Register | mfmsr | Guest Supervisor |
| Move to Machine State Register | mtmsr | Guest Supervisor |
| Write MSR External Enable | wrtee | Guest Supervisor |
| Write MSR External Enable Immediate | wrteei | Guest Supervisor |

**Table 31: No Information Flow or Externally Visible State Change Register Categories**

| Register Category | Table |
|---|---|
| General Purpose Registers | Appendix B, Table 67 |
| Floating-Point Registers | Appendix B, Table 68 |
| Branch Registers | Appendix B, Table 69 |
| Branch Control | Appendix B, Table 70 |
| Hardware Implementation Dependent Register | Appendix B, Table 71 |
| L1 Cache Registers | Appendix B, Table 72 |
| L2 Cache Registers | Appendix B, Table 73 |
| MMU Registers | Appendix B, Table 74 |
| Performance Monitoring Registers | Appendix B, Table 75 |

**Table 32: Process Control Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Machine State Register | MSR | | Guest Supervisor |
| Machine State Register Protect | MSRP | 311 | Hypervisor |
| Embedded Processor Control Register | EPCR | 307 | Hypervisor |
| Processor Version Register | PVR | 287 | Guest Supervisor RO |
| System Version Register | SVR | 1023 | Guest Supervisor RO |

The following bits are supported in the MSR:

- GS - Guest State. When set, the processor is running in guest mode.
- UCLE - User-mode Cache Lock Enabled. When set, user-mode cache locks do not make non-access violation exceptions.
- CE - Critical Interrupts. When set, critical input, processor doorbell critical, guest processor doorbell, and watchdog timer interrupt are enabled.
- EE - External Enabled. When set, external interrupts, decremented, fixed-interval timer, processor doorbell, guest processor doorbell, and embedded performance monitor interrupts are enabled.
- PR - User Mode. When set, the processor is running in user mode.
- FP - Floating-point available. When set, floating-point instructions are permitted to execute.
- ME - Machine check enable. When set, machine check interrupts are enabled.
- FE[01] - Floating-point exception mode. Freescale processor only implement precise exception mode.
- DE - Debug interrupt. When set, debug interrupts are enabled.
- IS - Instruction address space. When set, processor directs all instruction fetches to address space 1. Otherwise instruction fetches are directed to address space 0.
- DS - Data address space. When set, processor directs all data memory access to address space 1. Otherwise data memory access is directed to address space 0.
- Performance Monitoring Mark. When set, the process is marked for gather statistic data from.
- Recoverable Interrupt. When set, it is safe to return from a machine check, error report, or a non-maskable interrupt.

The MSRP register controls if the guest supervisor state is permitted to alter the UCLE, DE, and PMM bits in the MSR.

The EPCR offers limited interrupt direction control (i.e., are interrupts directed to the hypervisor state or to the guest supervisor state). The EPCR[DUVD] bit permits the disabling of debug events when running in the hypervisor mode. The EPCR[DGTMI] controls whether the TLB management instructions are available when running in guest supervisor mode. The EPCR[DMIUH] bit controls if the MAS registers are updated when running in the hypervisor mode.

The PVR and SVR registers provide read-only processor and SoC version values. Table 33 lists the registers provided in each processor core timing functions. The e500mc processor permits the Decrementer Auto-Reload Register (DECAR) to be read. This is a deviation from the architecture which defines the DECAR as Hypervisor WO.

Table 34 lists the registers provided in each processor to support interrupts.

**Table 33: Timer Registers**

| Register | Mnemonic | SPR # | Privilege |
|---|---|---|---|
| Time Control Register | TCR | 340 | Hypervisor |
| Timer Status Register | TSR | 336 | Hypervisor R/- |
| Time Base(R) | TBU,TBL | 268,26 | User RO |
| Time Base(W) | TBU,TBL | 284,28 | Hypervisor |
| Decrementer Register | DEC | 22 | Hypervisor |
| Decrementer Auto-Reload Register | DECAR | 54 | Hypervisor |
| Alternate Time Base Registers | ATBL,ATBU | 526, 527 | User RO |

**Table 34: Interrupt Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Standard Save/Restore Registers | SRR0, SRR1 | 26,27 | Guest |
| Critical Save/Restore Registers | CSRR0, | 58,59 | Hypervisor |
| Debug Save/Restore Registers | DSRR0, | 574,57 | Hypervisor |
| Machine Check Save/Restore Registers | MCSRR0, MC- | 570,57 1 | Hypervisor |
| Guest Save/Restore Registers | GSRR0, GSRR1 | 378,37 9 | Guest Supervisor |
| Data Exception Address Register | DEAR | 61 | Guest |
| Guest Data Exception Address Register | GDEAR | 381 | Guest |
| Interrupt Vector Prefix Register | IVPR | 63 | Hypervisor |
| Guest Interrupt Vector Prefix Register(R) | GIVPR | 447 | Guest Supervisor |
| Guest Interrupt Vector Prefix Register(RW) | GIVPR | 447 | Hypervisor |
| Critical input interrupt offset | IVOR0 | 400 | Hypervisor |
| Machine check interrupt offset | IVOR1 | 401 | Hypervisor |
| Data storage interrupt offset | IVOR2 | 402 | Hypervisor |
| Instruction storage interrupt offset | IVOR3 | 403 | Hypervisor |
| External input interrupt offset | IVOR4 | 404 | Hypervisor |
| Alignment interrupt offset | IVOR5 | 405 | Hypervisor |
| Program interrupt offset | IVOR6 | 406 | Hypervisor |
| Floating-point unavailable interrupt offset | IVOR7 | 407 | Hypervisor |
| System call interrupt offset | IVOR8 | 408 | Hypervisor |
| Auxiliary processor unavailable interrupt | IVOR9 | 409 | Hypervisor |
| Decrementer interrupt offset | IVOR10 | 410 | Hypervisor |
| Fixed-interval timer interrupt offset | IVOR11 | 411 | Hypervisor |
| Watchdog timer interrupt offset | IVOR12 | 412 | Hypervisor |
| Data TLB error interrupt offset | IVOR13 | 413 | Hypervisor |
| Instruction TLB error interrupt offset | IVOR14 | 414 | Hypervisor |
| Debug interrupt offset | IVOR15 | 415 | Hypervisor |

| | | | |
|---|---|---|---|
| Performance monitor interrupt offset | IVOR35 | 531 | Hypervisor |
| Processor doorbell interrupt offset | IVOR36 | 532 | Hypervisor |
| Processor doorbell critical interrupt offset | IVOR37 | 533 | Hypervisor |
| Guest processor doorbell interrupt offset | IVOR38 | 432 | Hypervisor |
| Guest processor doorbell critical and machine | IVOR39 | 433 | Hypervisor |
| Hypervisor system call interrupt offset | IVOR40 | 434 | Hypervisor |
| Hypervisor privilege interrupt offset | IVOR41 | 435 | Hypervisor |
| Guest data storage interrupt offset | GIVOR2 | 440 | Hypervisor |
| Guest instruction storage interrupt offset | GIVOR3 | 441 | Hypervisor |
| Guest external input interrupt offset | GIVOR4 | 442 | Hypervisor |
| Register | Mnemonic | SPR # | Privilege Level |
| Guest system call interrupt offset | GIVOR8 | 443 | Hypervisor |
| Guest data TLB error interrupt offset | GIVOR13 | 444 | Hypervisor |
| Guest instruction TLB error interrupt offset | GIVOR14 | 445 | Hypervisor |
| External proxy register | EPR | 702 | Guest |
| Guest external proxy register | GEPR | 380 | Guest |
| Exception syndrome register | ESR | 62 | Guest |
| Guest exception syndrome register | GESR | 383 | Guest |
| Processor ID register | PIR | 286 | Guest |
| Guest processor ID register | GPIR | 382 | Guest |
| Machine check address register | MACR | 573 | Hypervisor RO |
| Machine check address register upper | MACRU | 569 | Hypervisor RO |
| Machine check syndrome register | MCSR | 572 | Hypervisor |

When the processor core is running in guest-mode, several of the interrupt registers are automatically mapped to the corresponding guest registers. Table 35 defines the automatic mappings when the processor is running in guest-mode.

**Table 35: Guest-mode Mapped Interrupt Registers**

| Register Accessed | Mapped Register |
|---|---|
| SRR0 | GSRR0 |
| SRR1 | GSRR1 |
| EPR | GEPR |
| ESR | GESR |
| DEAR | GDEAR |
| PIR | GPIR |

Table 36 lists the software-use SPRs provided for each processor core. When the processor core is running in guest-mode, six of the software SPRs are automatically mapped to the corresponding guest register. Table 37 defines the automatic mappings when the processor is running in guest-mode.

The core device and control register (Table 38) provides an interface for the hypervisor to query and configure specific processor operations. This allows the hypervisor the ability to detect and take advantage of specific processor capabilities; as well as the option to disable some processor capabilities to assist in board layout and power management. The CDCSR0 register provides configurable support for the floating-point core device and instructions in the e500mc [Fre11a, Fre11b].

**Table 36: Software Use Special-Purpose Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| SPR general 0 | SPRG0 | 272 | Guest supervisor |
| SPR general 1 | SPRG1 | 273 | Guest supervisor |
| SPR general 2 | SPRG2 | 274 | Guest supervisor |
| SPR general 3(R) | SPRG3 | 259 | User RO |
| SPR general 3(RW) | SPRG3 | 275 | Guest supervisor |
| SPR general 4(R) | SPRG4 | 260 | User RO |
| SPR general 4(RW) | SPRG4 | 276 | Guest supervisor |
| SPR general 5(R) | SPRG5 | 261 | User RO |
| SPR general 5(RW) | SPRG5 | 277 | Guest supervisor |
| SPR general 6(R) | SPRG6 | 262 | User RO |
| SPR general 6(RW) | SPRG6 | 278 | Guest supervisor |
| SPR general 7(R) | SPRG7 | 263 | User RO |
| SPR general 7(RW) | SPRG7 | 279 | Guest supervisor |
| SPR general 8 | SPRG8 | 604 | Hypervisor |
| SPR general 9 | SPRG9 | 605 | Guest supervisor |
| Guest SPR general 0 | GSPRG0 | 368 | Guest supervisor |
| Guest SPR general 1 | GSPRG1 | 369 | Guest supervisor |
| Guest SPR general 2 | GSPRG2 | 370 | Guest supervisor |
| Guest SPR general 3 | GSPRG3 | 371 | Guest supervisor |
| User SPR general 0 | USPRG0 (VRSAVE) | 256 | User |

**Table 37: Guest-mode Mapped Software Use Special-Purpose Registers**

| Register Accessed | Mapped Register |
|---|---|
| SPGR0 | GSPGR0 |
| SPGR1 | GSPGR1 |
| SPGR2 | GSPGR2 |
| SPGR3(R) | GSPGR3 |
| SPGR3(RW) | GSPGR3 |

**Table 38: Core Device Control and Status Register**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Core device control and status register 0 | CDCSR0 | 696 | Hypervisor |

Table 39 lists the internal debug registers. The following internal debug registers are defined by Power ISA 2.06: DBCR3, IAC3, IAC4, DVC1, and DVC2, but are unimplemented in the e500mc processor [Fre11a].

**Table 39: Internal Debug Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Debug control register 0 | DBCR0 | 308 | Hypervisor |
| Debug control register 1 | DBCR1 | 309 | Hypervisor |
| Debug control register 2 | DBCR2 | 310 | Hypervisor |
| Debug control register 4 | DBCR4 | 563 | Hypervisor |
| Debug status register | DBSR | 304 | Hypervisor R/Clear |
| Debug status register write | DBSRWR | 306 | Hypervisor |
| Instruction address compare 1 | IAC1 | 312 | Hypervisor |
| Instruction address compare 2 | IAC2 | 313 | Hypervisor |
| Data address compare 1 | DAC1 | 316 | Hypervisor |
| Data address compare 2 | DAC2 | 317 | Hypervisor |
| Nexus SPR access configuration | NSPC | 984 | Hypervisor |
| Nexus SPR access data | NSPD | 983 | Hypervisor |
| Debug event | DEVENT | 975 | User |
| Debug data acquisition message | DDAM | 576 | User |
| Nexus processor ID register | NPIDR | 517 | User |

**Table 40: L1 Cache: Special Purpose Registers**

| Special Purpose Registers | Cache | Privilege Level |
|---|---|---|
| L1 Cache Configuration register 0 (L1CFG0) | Data | User RO |
| L1 Cache Configuration register 1 (L1CFG1) | Instruction | User RO |
| L1 Cache Control and Status Register 0 (L1SCR0) | Data | Hypervisor |
| L1 Cache Control and Status Register 1 (L1SCR1) | Instruction | Hypervisor |
| L1 Cache Control and Status Register 2 (L1SCR1) | Data | Hypervisor |

Level 1 Cache:
Each processor core has a private Level 1 (L1) instruction and data cache. Table 40 lists the five special-purpose registers used for the configuration, status, and control of these L1 caches.

The L1CFG0 and L1CFG1 registers provide read-only L1 cache configuration information to software. Both the instruction and data caches are non-blocking, physically addressed, and 32 Kbytes in size. The caches are organized as 64, eight-way sets of 64-byte cache lines. Both the caches provide parity, individual cache line locking, and implement a pseudo Least Recently Used (LRU) algorithm for cache line replacement.

The L1SCR0, L1SCR1, and L1SCR2 registers provide configuration and status for the L1 caches. The L1SCR0 and L1SCR1 registers control if the caches are: disabled or enabled, injecting cache faults, or reporting single-bit parity errors. These registers also provide status information useful for diagnostics. The L1SCR2 register provides additional control for the L1 data cache. Specifically, it configures the data cache write shadow and the data cache stash ID. Data cache write shadowing provides additional fault tolerance by using the Level 2 (L2) cache as a backup. If a failure occurs in the L1 cache, the data is recovered from the L2 cache. The data cache stash ID provides a cache target identifier to permit external devices to stash data in this processors L1 data cache. Data cache stash IDs between one and seven are illegal. Data cache stash ID of zero disables data cache stashing in this L1 data cache. Cache operations on the e500mc processor are never aborted. Cache lines are loaded in a single clock cycle. Any buffering required before the cache line load must be performed in the CoreNet interface.

L2 Cache:

Each processor core also has a private Level 2 (L2) backside cache. This cache is physically implemented as a dynamic Harvard architecture (i.e., a single, unified cache for both instructions and data, but provides Harvard architecture properties by providing different behaviors for the instruction and data cache entries). Instructions fetched into the L2 cache are marked with the N status bit to denote that the cache line was loaded incoherently.

Table 41 lists the special purpose registers used for the configuration, status, and control of the L2 cache.

**Table 41: L2 Cache: Special Purpose Registers**

| **Special Purpose Registers** | **Privilege Level** |
|---|---|
| L2 Cache Configuration register 0 (L2CFG0) | User RO |
| L2 Cache Control and Status Register 0 (L2SCR0) | Hypervisor |
| L2 Cache Control and Status Register 1 (L2SCR1) | Hypervisor |
| L2 Cache Error Disable Register (L2ERRDIS) | Hypervisor |
| L2 Cache Error Detect Register (L2ERRDET) | Hypervisor |
| L2 Cache Error Interrupt Enable Register (L2ERRINTEN) | Hypervisor |
| L2 Cache Error Control Register (L2ERRCTL) | Hypervisor |
| L2 Cache Error Address Capture Registers (L2ERRADDR, L2ERREADDR) | Hypervisor |
| L2 Cache Error Capture Data Registers (L2ERRDATALO, L2ERREDATAHI) | Hypervisor |
| L2 Cache Error Capture ECC Syndrome Register (L2CAPTECC) | Hypervisor |
| L2 Cache Error Attribute Register (L2ERRATTR) | Hypervisor |
| L2 Cache Error Injection Control Register (L2ERRINJCTL) | Hypervisor |
| L2 Cache Error Injection Mask Registers (L2ERRINJLO, L2ERRINJHI) | Hypervisor |

The L2CFG0 register provides the read-only organization and capabilities for the L2 cache to software. The L2 cache is 128 Kbytes in size and is organized as 256, eight-way sets of 64-byte cache lines. This cache provides parity L2 cache tag error handling, parity and ECC L2 cache data error handling, instruction and data partitioning, individual cache line locking, and a choice of pseudo-LRU, streaming PLRU, or streaming PLRU with the aging cache line replacement algorithms.

The L2CSR0 and L2CSR1 registers provide general control and status information for the L2 cache. The L2CSR0 register controls if the cache is disabled or enabled, if ECC error checking is enabled or disabled, the L2 cache line replacement algorithm, and various cache flushing methods. The L2CSR0 also provides cache partitioning support. Cache partitioning specifies how many ways should be used for instructions and how many ways should be used for data.

The L2CSR1 register controls L2 instruction loss limits and L2 snoop win limits. The L2CSR also provides bits to store the data cache stash ID. Similar to the L1 data cache, data stash ids between one and seven are illegal and a data stash ID of zero disables stashing in the L2 cache.

The L2 Cache error registers provide error detection and reporting. The L2ERRINJCTL register permits the injection of ECC and parity errors into the L2 cache for testing purposes [Fre11a].

Cache Stashing:
The P4080 architecture supports peripheral devices to preload (i.e., stash) data into the L1 data and/or the unified L2 cache. Cache stashing is intended as a performance hint from the peripheral by pre-populating the caches with data before the processor core needs to request it. The processor core is a passive recipient of the cache stashing operation [Fre11a].

CoreNet Bus Interface Unit:
The CoreNet bus interface provides the primary interface from each processor core to the reset of the P4080 architecture.

Interrupts:
Interrupts cause the processor to alter the program flow by: (1) saving the next instruction address and current processor context information, (2) beginning the execution of the interrupt handler at a pre-determined address location, and (3) resuming program flow by restoring the context information and executing the save instruction address. In the P4080 multicore architecture, the Multicore Programmable Interrupt Controller (MPIC) manages the routing and delivery of interrupts from SoC components to the processor cores.

Table 42 provides the Interrupt Vector Offset Register (IVOR) (i.e., the interrupt number) and the description for the e500mc interrupts [Fre11a].

#### 6.2.1.2 Safeguards

Cache Stashing:
Cache stashing can be prohibited in the L1 data cache by configuring the Data Cache Stash ID value in the L1 Cache Control and Status Register 2 (L1CSR2) register to 0. Cache stashing in the unified L2 cache can be prohibited by configuring the L2 Cache Stash ID value in the L2 Cache Control and Status Register 1 (L2CSR1) register to 0. Both the L1CSR2 and L2CSR1 special-purpose registers require hypervisor privileges to read and write.

Alternatively, clearing the DCE bit in the L1 Cache Control and Status Register 0 (L1CSR0) register and clearing the L2E bit in the L2 Cache Control and Status Register 0 (L2CSR0) register will disable both the L1 data cache and the unified L2 cache [Fre11a].

Table 42: P4080 All IVORs

| IVOR | Interrupt |
|---|---|
| IVOR0 | Critical Input |
| IVOR1 | Machine Check, Error Report |
| IVOR2,GIVOR2 | Data Storage (DSI) |
| IVOR3,GIVOR3 | Instruction Storage (ISI) |
| IVOR4,GIVOR4 | External Interrupt |
| IVOR5 | Alignment |
| IVOR6 | Program |
| IVOR7 | Floating-Point Unavailable |
| IVOR8,GIVOR | System Call |
| IVOR10 | Decrementer |
| IVOR11 | Fixed Interval Timer |
| IVOR12 | Watchdog |
| IVOR13,GIVOR13 | Data TLB Error |
| IVOR14,GIVOR13 | Instruction TLB Error |
| IVOR15 | Debug |
| IVOR35 | Performance Monitor |
| IVOR36 | Processor Doorbell |
| IVOR37 | Processor Doorbell Critical |
| IVOR38 | Guest Processor Doorbell |
| IVOR39 | Guest Processor Doorbell Critical/ Doorbell Machine Check |
| IVOR40 | Hypervisor System Call |
| IVOR41 | Hypervisor Privilege |

Table 43: P4080 Maskable IVORs

| **IVOR** | Enabled by |
|---|---|
| IVOR0 | MSR[CE] or MSR[GS] |
| IVOR1 | MSR[ME] or MSR[GS] |
| IVOR4,GIVOR4 | MSR[EE] or MSR[GS] |
| IVOR6 | Floating Point Exception: MSR[FE0][FE1] |
| IVOR10 | (MSR[EE] or MSR[GS]) and TCR[DIE] |
| IVOR11 | (MSR[EE] or MSR[GS]) and TCR[FIE] |
| IVOR12 | (MSR[EE] or MSR[GS]) and TCR[WIE] |
| IVOR15 | MSR[DE] and DBCR0[IDM] |
| IVOR35 | MSR[EE] or MSR[GS] |
| IVOR36 | MSR[EE] or MSR[GS] |
| IVOR37 | MSR[CE] or MSR[GS] |
| IVOR38 | MSR[EE] and MSR[GS] |
| IVOR39 | Critical: (MSR[CE] and MSR[GS] Machine Check: MSR[ME] and MSR[GS] |

Interrupts:

The P4080 multicore architecture supports interrupt masking in both the e500mc processor core and in the MPIC. Table 43 lists which IVORs are maskable and how to mask them.

Disable Component:
The CONFIG_COREDISR register provides eight bits to represent the state of each processor core. Each processor core can individually be disabled by writing a value of one to the appropriate bit in the CONFIG_COREDISR register.

### 6.2.2 CoreNet Coherency Fabric

This section provides the information flow analysis of the CoreNet described in Section 6.1.2.

#### 6.2.2.1 Information Flows

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

The CoreNet coherency fabric is the primary communication bus for the P4080 multicore architecture. The CoreNet coherency fabric provides 32 address ranges called Local Access Windows (LAWs). All system addresses, with the exception of the boot window and configuration space mapped by the CCSRBAR register, must be mapped by a LAW. The lowest-numbered LAW takes precedence when addresses are mapped to multiple LAWs [Fre11c].

Bus Transactions:
CoreNet coherency fabric bus transactions provide information flows to every component in the system.

CoreNet Coherency Cache:
CoreNet transactions, with proper destination addresses, breach the abstract polyhedron surrounding the CoreNet coherency cache. Information flows may pass through the CoreNet coherency fabric to the DDR memory controllers if the CoreNet transaction is a write or when the address is not present in the CoreNet coherency cache.

Interrupts:
The CoreNet coherency fabric components provide two interrupts. The first interrupt (8) is dedicated to PAMU access violations. The second interrupt is shared by multiple components. The CoreNet coherency fabric, CoreNet coherency cache, and PAMUs generate this shared interrupt when internal error conditions occur.

#### 6.2.2.2 Safeguards

Peripheral Access Management Unit:
Each PAMU is configured to reject all transactions by default. The LIODN must be configured for each master capable device behind the PAMU. In normal mode, the hypervisor must configure each memory regions in the PAMU with appropriate permissions.

The PAMUx_PC (where x is the PAMU number) provides a PE (PAMU Enabled) bit and PGC (PAMU Gate Check) bit. When the PGC bit is set, the PAMU blocks all peripheral accesses. When the PGC bit is set and the PE is cleared, the PAMU also blocks all peripheral accesses. When the PGC and PE bits are both set, then the PAMU performs the authorization and translation functions.

The DCFG_PAMUBYPENR register contains control bits for enabling the PAMU bypass mode for each PAMU.

### 6.2.3 CoreNet Coherency Cache

This section provides the information flow analysis of the CoreNe0 coherency cache described in Section 6.1.3.

#### 6.2.3.1 Safeguards

Although, the CoreNet platform cache provides no ability to restrict information flows entering, leaving, or going through it; it does provide a safeguard for the cache replacement policy. When a cache miss occurs, the CoreNet platform will look up by partition, action, and way to determine the appropriate cache replacement policy. The CoreNet platform cache can also be disabled, thus defining two externally visible states: enabled and disabled.

### 6.2.4 Enhanced Local Bus Controller

This section provides the information flow analysis of the bus controller from Section 6.1.4

#### 6.2.4.1 Information Flows

Due to the coupling between the attached peripherals, the abstract polyhedron completely encloses the eLBC and the attached peripherals.

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

<u>Bus Transactions:</u>
Local bus transactions are the primary information flow through the enhanced Local Bus Controller (eLBC). Local bus transactions may be initiated either from some device connected to the CoreNet coherency fabric or by another peripheral connected to the eLBC.

<u>Interrupts:</u>
The eLBC provides two interrupt lines. The first interrupt (9) is dedicated to the eLBC. This interrupt multiplexes the interrupt lines for the miscellaneous peripheral devices. The second interrupt is shared by multiple components. The eLBC may generate this interrupt when internal error conditions occur.

Boot Device:
The P4080 architecture supports booting from the flash that is attached to the FCM peripheral interface controller.

#### 6.2.4.2 Safeguards

Interrupts:
Both interrupt lines are routed to the Multicore Programmable Interrupt Controller. The MPIC permits both of these interrupts to be masked independently. If masking is not desired, the MPIC also provides configurable options for priority and processor core destinations for both interrupts.

Masking the shared interrupt at the MPIC may be heavy handed. The eLBC provides fine-grained control over the eLBC-generated interrupts via the eLBC_LTEIR register. The eLBC does not provide fine-grained control for interrupt 9 used by the eLBC peripherals.

Boot Device:
The P4080 multicore architecture provides a trust architecture (consisting of hardware and software) to provide a secure boot environment. At the time of writing, not all of the documentation for the trust architecture was publicly released from Freescale and the implementation details are beyond the scope of this project.

Disable Component:
Setting bit 19 in the CONFIG_DEVDISR1 register will disable the eLBC component. Setting bit 29 and/or 30 in the DCFG_ELBCCLKDR register will disable one or both of the eLBC clocks.

Components on the eLBC bus can be disabled by setting the appropriate bits in the CONFIG_DEVDISR1 register.

### 6.2.5 Data Path Acceleration Architecture: Individual Components

Initial framework analysis treats each component in the DPAA (i.e., BM, QM, FM, SEC, and PME) as individual system-high compartments. Each component is fully enclosed in its own abstract polyhedron. Information flows breaching the abstract polyhedron or externally visible events require mitigating safeguards. However, as the following analysis of BM demonstrates, there are insufficient safeguards in place when the DPAA component is treated (i.e., analyzed) as a set of individual components. For brevity in this report, only the BM component is shown to have inadequate information flow safeguards, although the other DPAA components suffer the same deficits. In contrast, Section 6.2.6 provides a complete information flow and safeguard analysis for the DPAA as a whole (i.e., an alternative polyhedron encompassing all DPAA components).

#### 6.2.5.1 Information Flows

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.

3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

Buffer Manager:
The BM provides 11 interrupt lines that are routed to the MPIC. Ten of these interrupt lines are provided to alert the software to the fact that one of the software portals needs attention. The last interrupt line provides generic error information for the BM.

The BM does not directly communicate with the CoreNet coherency fabric. All BM CoreNet transactions are mediated by the QM. Software portal registers are memory mapped into one or more CoreNet LAWs, providing the interface for the processor cores to interface with the 10 software portals.

A major information flow that breaches the abstract polyhedron is the direct connect portals. The P4080 DPAA implementation provides four direct connect portals: one to each FM, one to the SE, and one to the PME.

A major information flow breaching the abstract polyhedron is the software portals. The P4080 DPAA implementation provides 10 software portals: one for use by each processor core, one for use by the hypervisor, and one for use by an external processor accessing the QM. The processor cores, hypervisor, and external processor all invoke CoreNet transactions to interact with the software portals. The BM does not have direct access to the CoreNet coherency fabric. All CoreNet access to the BM is mediated through the QM.

A potential covert information flow exists in the software portal query command. This command can be issued by any software portal and will return the status (i.e., are free buffers available in the pool) for all 64 buffer pools.

A potential covert information flow exists in the buffer pools. While the software portals may provide atomic access to the buffers in the buffer pools, the BM does not perform any sanity checks on the buffers. This includes ensuring that a memory buffer has been properly sanitized before re-assigning it to software or direct connect portal.

#### 6.2.5.2 Safeguards

Buffer Manager:
The BM component provides the ability to enable, to disable, and to inhibit the generation of interrupts. The MPIC also provides methods for masking or routing interrupts to the specified processor core(s).

The BM component was designed using the QM to mediate access to the CoreNet coherency fabric. The DPAA does not provide a method for the BM to communicate to its private, external memory region without involving the QM component. **No safeguard exists to manage this information flow**.

The DPAA was designed to provide low-latency, direct connect portals for hardware components. The DPAA does not provide alternative methods for on-chip hardware components to access BM. **No safeguard exists to manage this information flow**.

The P4080 DPAA software portals were designed to be isolated to a single software unit. The memory sizes and alignments are setup so the processor MMUs and CoreNet LAWs can be programmed by the hypervisor to provide this proper isolation.

The DPAA is designed to provide atomic access to buffer pools. It was not designed to support confidentiality. The 64 buffer pools are a shared resource to all the software and direct connect portals. Software portals can alter the number of free buffers in any of the pools by acquiring or releasing buffers. The portal query command returns the status of all 64 buffer pools in a single command. **No safeguard exists to manage this information flow**.

### 6.2.6 Data Path Acceleration Architecture: Grouped Components

In the previous section, it was shown that DPAA components, when analyzed separately, do not have sufficient information flow safeguards for high assurance systems. However, when the DPAA is analyzed as a whole (i.e., abstract polyhedron fully encloses the BM, the QM, both FMs, the PME, the SEC, and one or more processor cores) it can be shown the P4080 multicore architecture provides adequate information flow safeguards.

#### 6.2.6.1 Information Flows

In this configuration, the processor core(s) manage the DPAA components and proxy the data to other parts of the system. The processor cores would provide MLS guard functionality for the DPAA components to the rest of the system. Information flows breaching this abstract polyhedron or changing the color of the abstract polyhedron are examined in additional detail below.

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

<u>Processor Core(s):</u>
The processor cores were analyzed individually in Section 6.2.1. The same analysis applies to the processor cores in this configuration as well.

<u>CoreNet:</u>
Several of the DPAA components require private, external memory regions in system memory. These memory regions need to be properly protected by CoreNet LAWs and the hypervisor needs to configure processor cores and the PAMUs to ensure these private, external memory regions are not accessible to other components.

Interrupts:
The DPAA implementation in the P4080 multicore architecture supports multiple interrupts in each component. These interrupts can be roughly classified as normal (e.g., data is available for the processor) or when errors occur in the components (e.g., ECC memory error).

SerDes:
FM interfaces with an external PHY or SerDes device to complete the interface to the physical layer.

#### 6.2.6.2 Safeguards

Processor Core(s):
The processor cores were analyzed individually in Section 6.2.1. The same analysis applies to the processor cores in this configuration as well.

CoreNet:
The hypervisor must configure the processor core MMUs and the CoreNet PAMUs to ensure that the private, external memory regions required for the DPAA are not accessible to other components. The hypervisor must configure the non-grouped processor core MMUs to ensure they do not have access to any of the DPAA components. The hypervisor must also configure the PAMUs guarding DPAA components to only communicate with specified memory regions and the grouped processor cores.

Additionally, the P4080 multicore architecture supports logical partitions. Placing all the DPAA grouped components into a logical partition would allow the hardware to provide additional checks to ensure that the above configuration was implemented properly.

Interrupts:
All the DPAA interrupt lines are routed to the MPIC. The MPIC can be configured to either mask these interrupts or to route them to the grouped processor cores.

SerDes:
SerDes is a point-to-point serial bus. The SerDes lanes used for connecting FM to the physical layer interface are private. No additional safeguards needed.

Disable Components:
Table 44 provides the bit indexes in the DCFG_DEVDISR2 register used to disable the individual DPAA components.

### 6.2.7 On-Chip Network: Individual Components

Initial framework analysis treats each component in the On-Chip Network (OCN) (i.e., PCIe, RapidIO, sRIO, and DMA controllers) as individual system-high compartments. Each component is fully enclosed in its own abstract polyhedron. Information flows breaching the abstract polyhedron or externally visible events require mitigating safeguards. However, as the following analysis of the PCIe demonstrates, there are insufficient safeguards in place when the OCN components are treated (i.e., analyzed) as a set of individual components. For brevity in this report, only the PCIe component is shown to have inadequate information flow safeguards, although the other OCN components suffer the same deficits. In contrast, Section 6.2.7 provides a complete information flow and safeguard analysis for the OCN as a whole (i.e., an alternative polyhedron encompassing all OCN components).

**Table 44: DCFG_DEVDISR2 Component Disable Bits**

| Component | Bit(s) |
|---|---|
| PME | 0 |
| SEC | 1 |
| QM and BM | 4 |
| FM 1 | 6 |
| FM 1 - 10GE | 7 |
| FM 1 - 4 x 1GE | 8, 9, 10, and 11 |
| FM 2 | 14 |
| FM 2 - 10GE | 15 |
| FM 2 - 4 x 1GE | 16, 17, 18, and 19 |

### 6.2.7.1 Information Flows

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

Peripheral Component Interconnect Express:
Each PCIe controller provides two interrupt lines that are routed to the MPIC. One of the interrupts is dedicated and alerts software when the PCIe needs attention. The other interrupt is shared and provides generic error information for the PCIe controller.

The PCIe controllers connect to a common crossbar switch. This crossbar switch provides mesh communications to the three PCIe controllers, DMA controllers, RapidIO Message Unit, sRIO controllers, and the CoreNet coherency fabric. The crossbar switch provides no access controls, permitting one PCIe controller talking to a different PCIe controller without PAMU access control checks.

### 6.2.7.2 Safeguards

Peripheral Component Interconnect Express:
PCIe provides registers to enable, disable, and inhibit the generation of interrupts. The MPIC also provides methods for masking and routing interrupts to specific processor cores.

The common crossbar switch does not provide safeguards to restrict access between the OCN components. The PAMU only applies access controls for inbound data for the CoreNet coherency fabric. **No safeguard exists to manage this information flow.**

### 6.2.8 On-Chip Network: Grouped Components

In the previous section, it was shown that the OCN components, when analyzed separately, do not have sufficient information flow safeguards for high-assurance systems. However, when the OCN is analyzed as a whole (i.e., abstract polyhedron fully encloses the PCIe controllers, DMA controllers, RapidIO Message Unit, and sRIO controllers) it can be shown that the P4080 multicore architecture provides adequate information flow safeguards.

#### 6.2.8.1 Information Flows

To identify information flows, each processor core is enclosed in an abstract polyhedron. The surface color of the polyhedron represents the externally visible state. Using this framework:

1. Any information flow breaching any surface of the polyhedron must be evaluated.
2. Any information flow altering the externally visible state must be evaluated.
3. Any information flow not breaching the polyhedron nor modifying the externally visible state can be ignored.

There are three information flows that breach the polyhedron surface: interrupts, CoreNet transactions, and SerDes transactions.

#### 6.2.8.2 Safeguards

The PAMU provides protection for the CoreNet transactions. There is a special notice for the configuration of DMA in multi-partition systems.

The DMA controller uses a privileged copy of the CoreNet LAWs to resolve source and destination targets. The P4080 reference manual [Fre11c] specifies two methods for ensuring DMA works properly in multi-partition systems: 1) virtualize the Operating System (OS) to ensure that hypervisor calls are made to program the DMA controllers, or 2) ensure that all logical address are mapped one-to-one to physical addresses and that guest I/O logical addresses are mapped into the same globally unique system-wide range, which does not overlap with other non-I/O addresses.

SerDes is a point-to-point serial bus. The SerDes lanes used for connecting PCIe and sRIO to external interfaces are private. No additional safeguards are needed.

Disable Component:
The three ePCI controllers can be disabled by setting bits 0, 1, and/or 2 in the DCFG_DEVDISR1 register. The RapidIO message unit can be disabled by setting bit 4 in the DCFG_DEVDISR1 register. The two sRIO controllers can be disabled by setting bits 5 and/or 6 in the DCFG_DEVDISR1 register. The two DMA controllers can be disabled by setting bits 9, and/or 10 in the DCFG_DEVDISR1 register.

### 6.2.9 Real Time Debug

This section provides the information flow analysis of the real-time debug unit described in Section 6.1.7

#### 6.2.9.1 Information Flows

Without access to the *Advanced QorIQ Debug and Performance Monitoring Guide*, it was not possible to identify all of the information flows. For the purpose of this report, the worst case was assumed where both storage and timing covert channels exist via shared debug components. Since most of functional components provide debug hooks, the real time debug component provides information flows to all components.

#### 6.2.9.2 Safeguards

<u>Debug Control and Status Registers:</u>
The Debug Control and Status Registers (DCSR) allow software access to the hardware debug and performance monitoring capabilities built into the P4080 multicore architecture. There are two methods for accessing the DSCR registers: (1) using an external debug controller and (2) by creating a CoreNet LAW and mapping this address space into system memory space.

An external debug controller requires physical access to the device. One solution to protecting the device against physical attacks would be to encapsulate the circuitry in a hard epoxy material suitable for tamper detection per Physical Security Level 3 defined by FIPS 140-2 [Dep01].

The DCSR memory space would only be accessible to software if it was mapped in by a CoreNet LAW. The hypervisor should ensure this memory space is not accessible.

<u>Disable Component:</u>
Setting bit 15 in the CONFIG_DEVDISR1 register will disable the debug controller. Setting bit 16 in the CONFIG_DEVDISR1 register will disable the Nexus/Aurora link layer.

## 6.3 APPLYING THE SECURITY POLICY

The final step in the framework is to apply the Security Policy to the information flows identified in the previous step.

As described in the previous chapter, red networks are cleared to transmit classified and/or sensitive information in plaintext. Black networks are cleared to transmit encrypted classified and/or sensitive information. Figure 16 illustrates how two security gateways would be used to transmit classified and/or sensitive information between two red networks via a black network.

In this scenario, a paper analysis of the Freescale P4080 multicore architecture is used. One primary advantage of the Freescale P4080 multicore architecture is the high-speed DPAA architecture. Red networks would be connected using the 1GE network interfaces. The 10GE network interfaces would then be used to transmit data over the black networks. The DPAA architecture could be configured to send red network data to a specific, isolated processor core that would encrypt the data using the correct encryption key. This data would then be rerouted back through the DPAA to traverse the black network. Incoming black-network data would use the parsing capabilities of the FM to identify the target red network. This data would then be routed to the appropriate isolated processor core to decrypt the network data and then transmit the data to the correct red network.

The security policy used for this report:

1. Red network data is only permitted to be transmitted to a red network with the same classification. Separate cryptographic keys shall be used to ensure the separation of data at different classification levels.
2. All encrypted data must pass through the guard before transmission on the black network.

## 6.4 CONCLUSIONS

Based on the analysis provided in this chapter, we cannot recommend the Freescale P4080 multicore architecture as a generic MILS multicore architecture for the following reasons:

1. The coupling of the Data Path Acceleration Architecture (DPAA) components required the abstract polyhedron to fully enclose all the DPAA components plus at least one processor core. This component coupling dramatically reduces the effectiveness of the Freescale P4080 multicore architecture for MILS applications in high-assurance communication systems.
2. The coupling of Enhanced Local Bus Controller (eLBC) peripherals required the abstract polyhedron to fully enclose the eLBC and the attached peripherals.
3. The coupling of the On-Chip Network (OCN) required the abstract polyhedron to fully enclose all the OCN and the attached peripherals.

# 7 EVALUATING INTEL NEHALEM ARCHITECTURE USING THE FRAMEWORK

The Intel x86 processor family was started with the release of the 8086 processor in 1978. Since that processor release, Intel has evolved the processor architecture through many technical evolutions, while still maintaining backwards compatibility. Object code created for the 8086/8088, 16-bit, processors will still run on the latest processor in the Intel 64 and IA-32 architecture families [Int12].

The x86 processor family provides four native privilege levels: ring 0, ring 1, ring 2, and ring 3. Ring 0 is the most privileged and is typically where the operating system kernel runs. Ring 3 is the least privileged and is typically where applications are run. For backward compatibility reasons, these processors do not implement a separate privilege level for the hypervisor. Instead, Intel added the Virtual Machine Extensions (VMX) for virtualization support. The VMX extensions provide two software modes: hypervisor (i.e., VMX root) and guest system (i.e., VMX non-root). After guest configuration, the hypervisor transitions to the guest system mode by performing a VM-entry. While executing in guest system mode, the processor core runs with all four native privilege levels, but will VM-exit to the hypervisor when specific conditions are hit. The hypervisor mode, added by the VMX extensions, creates a new pseudo-privilege level (commonly called ring -1).

In addition to the above privilege levels, the processor also provides an extremely privileged System Management Mode (SMM) (commonly called ring -2). SMM handles system-wide functions like power management, system hardware control, or proprietary Original Equipment Manufacturer (OEM)-designed code. SMM is intended for use by system firmware, not by hypervisors, operating systems, or application software. When a System Management Interrupt (SMI) is received, the processor saves the context of the current task, disables the interrupts, and changes to the SMRAM address space before executing the SMM code. The SMRAM address space is similar to the real-address mode since there are no privilege levels or address mappings. This allows SMM to execute all system instructions, access all I/O addresses, and access all system memory. The RSM instruction is only executable in SMM and is used to return control back to the interrupted task [Int12].

This analysis assumes that a trusted hypervisor has initialized the processor cores, configured each processor core in protected mode, and initialized the VMX extensions. VMX root operations (i.e., hypervisor) are managed by this trusted hypervisor. This analysis focuses on information flows between guest operating systems running on different physical processor cores. Figure 26 illustrates the different privilege levels present in each processor core.

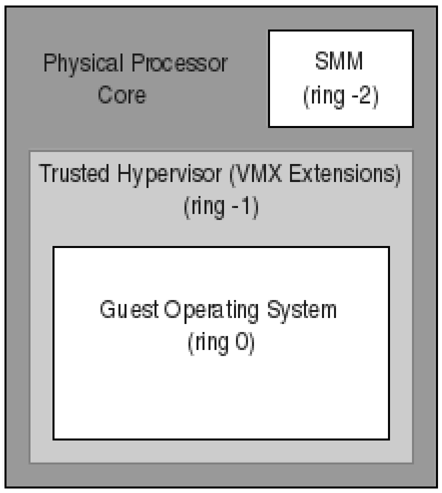

Physical Processor Core
SMM (ring -2)
Trusted Hypervisor (VMX Extensions) (ring -1)
Guest Operating System (ring 0)

**Figure 26: Processor Privilege Levels**

Due to the longevity and popularity of the x86 processor family, this processor family has received attention from security researchers. For brevity in this report, a non-exhaustive review of a few vulnerabilities targeting interrupts, VMX instructions, TXT-based trusted boot, and SMM mode are discussed. In this chapter, the Intel Nehalem multicore architecture is analyzed to see if the information flow and safeguard analysis presented in the framework could identify additional vulnerabilities.

## 7.1 IDENTIFY HARDWARE COMPONENTS

The first step in the framework is to identify the hardware components.

For brevity in this report, the framework was only applied to a small portion of the architecture. Specifically, the analysis focused on information flows and safeguards for the system initialization process and interprocessor interrupts.

### 7.1.1 Processor Cores

Guest OS Initialization:
On power-up or RESET, the hardware dynamically selects one processor core on the system bus and designates it as the BootStrap Processor (BSP). The other processor cores are designated as Application Processors (AP). The BSP initializes the system by executing the BIOS bootstrap code to configure the Advanced Programmable Interrupt Controllers (APIC), initialize system-wide data structures, and then completes the initialization of the other APs. Once this initialization is complete, the BSP begins the initialization of the trusted hypervisor code [Int12].

One of the first tasks for the trusted hypervisor is to load the microcode updates into each of the processing cores[8]. Next, the trusted hypervisor completes the initialization and setup of the processing environment. Once this initialization is completed, the trusted hypervisor issues the

[8] Many Intel processors provide the capability to correct processor errata by loading an Intel-supplied data block (i.e., the microcode update) into the processor.

VMXON instruction on each processor core. Now the trusted hypervisor configures the guest environment on each processor core using the VMX instructions. The VMLAUNCH instruction is executed on each processor core, telling it to begin running the guest operating system [Int12]..

Interprocessor Interrupts:
The InterProcessor Interrupts (IPI) mechanism is used to maintain synchronization between the processor cores. The IPI mechanism is used to interrupt individual or groups of processor cores connected to the system bus. Each processor core contains a local Advanced Programmable Interrupt Controller (APIC) for managing interrupts. The APIC registers are memory mapped into a 4-Kbyte region of the processor core's physical memory space. One of the registers in this memory region is the Interrupt Command Register (ICR). The ICR is presented as two 32-bit registers (ICR_LOW and ICR_HIGH) to form a 64-bit register. Figure 27 shows the ICR register format. The ICR is both readable and writeable by the guest operating system [Int12].

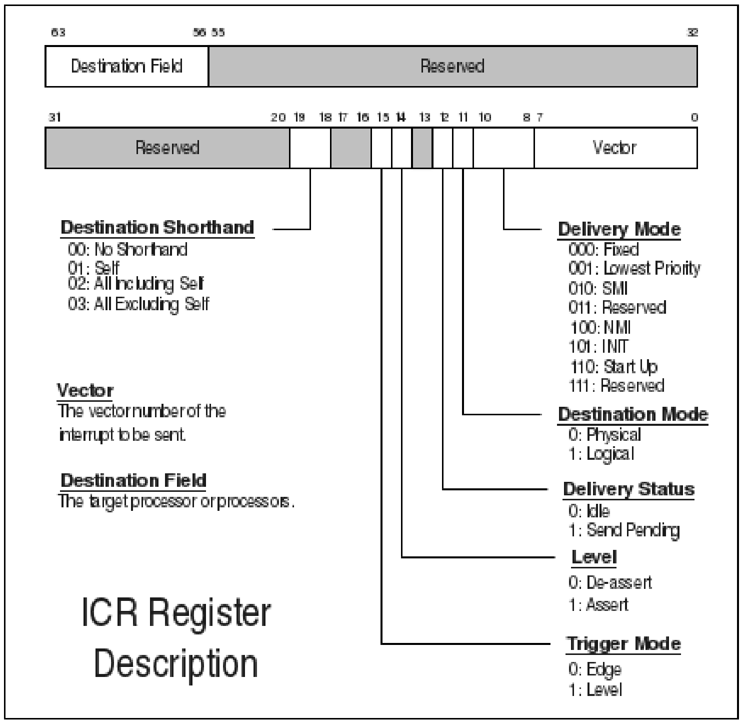


**Figure 27: ICR Register**

## 7.2 IDENTIFY AND EVALUATE INFORMATION FLOWS

The second step of the framework is to identify information flows and safeguards.

### 7.2.1 Processor Cores

To identify information flows between the guest operating systems, only the guest operating system in each processor core is completely enclosed in the abstract polyhedron. The non-enclosed portion of each processor core is the control of the trusted hypervisor and not analyzed as part of this analysis. Three types of information flows exist in the system:

1. Information flows generated by the trusted hypervisor that may or may not breach any abstract polyhedrons.
2. Information flows generated from an abstract polyhedron that does not affect the state or breach any other abstract polyhedron.
3. Information flows generated from an abstract polyhedron which do effect the state of or breach at least one other abstract polyhedron.

Only type 3 information flows meet the definition of a covert communication channel. Information flows not meeting the requirements for type 3 can be ignored as part of the analysis.

#### 7.2.1.1 Information Flows

Interprocessor Interrupts:
Guest operating systems can generate interprocessor interrupts by writing to the ICR in the local APIC. The ICR permits the guest operating system to specify the destination (including interrupt broadcast options), the interrupt vector, and the delivery mode. Specifically, the ICR permits the guest operating system to generate a Non-Maskable Interrupt (NMI).

A potential covert information flow exists when using interprocessor interrupts. A guest operating system, enclosed in the polyhedron, can generate interrupts for other processor cores by writing to the ICR. The VMX instructions provide methods for directing an interrupt to either the trusted hypervisor or to the guest operating system. A direct communication channel exists when the trusted hypervisor configures the interrupt to be delivered to the guest operating system on the target processor cores. An indirect communication channel also exists when the trusted hypervisor handles the interrupt on the target processor cores. This indirect communication channel exists given the fact that the interrupt will cause a VM-exit to relinquish control to the hypervisor to handle the interrupt. This VM-exit takes a measurable amount of time, thus providing an indirect timing communication channel.

The Startup InterProcessor Interrupt Attack:
As part of the multicore boot process, two interrupts are used. The INIT interrupt resets the processor and puts the processor in the wait-for-SIPI state. When the Startup InterProcessor Interrupt (SIPI) is sent, it directs the processor to start code execution at a physical memory address. [Int12, WRd]. The only documented method for generating the SIPI is via the ICR. However, in 2011, researchers discovered a method for generating SIPI interrupts from PCIe devices [Int12].

This attack is possible since these processor cores also support the Message Signaled Interrupts (MSI). MSI were introduced as part of the PCI Local Bus Specification, Rev 2.2. Figure 28 describes the layout of the MSI Message Data Register (MDR).

The researchers in [WRd] recognized the similarity between the ICR (Figure 27) and the MSI MDR (Figure 28) formats. Specifically, they recognized that the SIPI was initiated in the ICR by using the *Start Up* (i.e., 110b) value. In the MSI MDR, this value was reserved. The researchers crafted a MSI message. Figure 29 illustrates the attack.

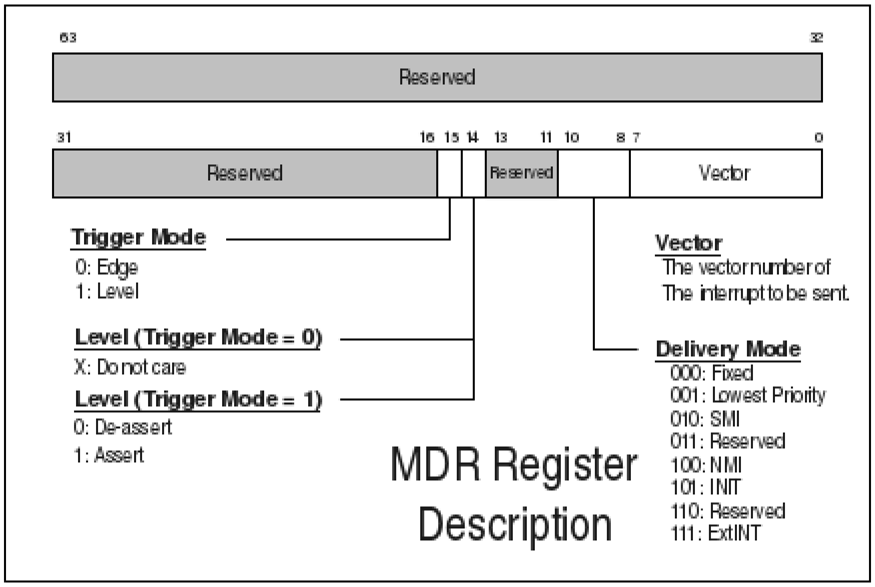


**Figure 28: MDR Register Format**

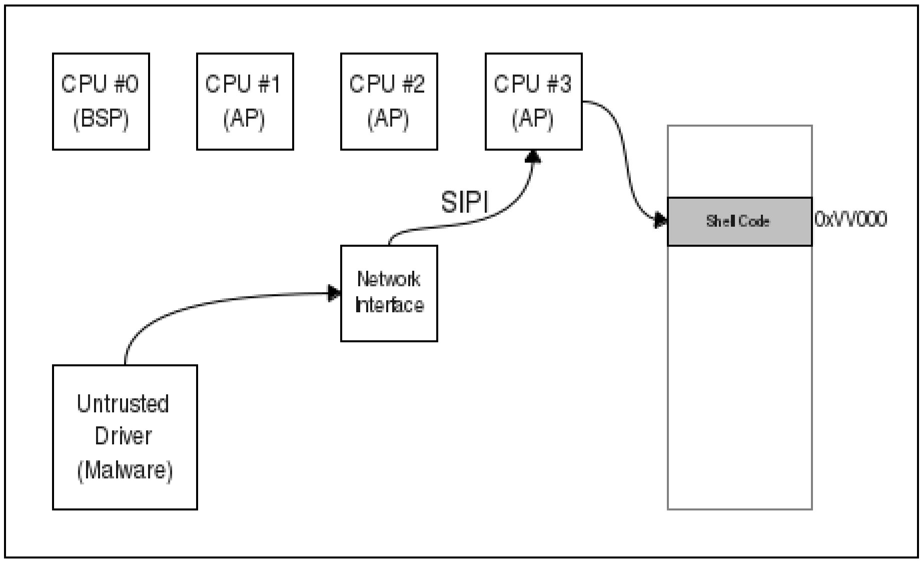


**Figure 29: SIPI Attack**

The difficulty of this attack is that the shell code must be placed in physical memory below 1-MByte and must start on a page boundary (0x1000) [WRd].

System Management Mode:

SMM is the most privileged mode on the system. This mode is not intended to be used for software applications; it offers no privilege levels and circumvents all protections put into place by the VMX instructions, page table permissions, ring levels, Etc.

#### 7.2.1.2 Safeguards

The VMX extensions implement a Virtual-Machine Control Data Structure (VMCS) to manage transitions from the trusted hypervisor to the guest operating systems. The VMCS data is organized into six logical groups: (1) Guest-state area, (2) Host-state area, (3) VM-execution control fields, (4) VM-exit control fields, (5) VM-entry control fields, and (6) VM-exit information fields. The VM-execution control fields control the processor behavior when operating in the guest operating system. These fields partially control what actions will cause a VM-exit back to the trusted hypervisor.

The VM-execution control fields are composed of two 32-bit values. The first 32-bit value contains the primary processor-based VM-execution controls and the second 32-bit value contains the secondary processor-based VM-execution controls. Table 45 identifies the bits in the VM-execution control fields that must be set in order to force the guest operating system to VM-exit when accessing the physical pages for the local ASIC.

**Table 45: VM-execution Controls for local ASIC Access Protection**

| | **Bit** | **Description** |
|---|---|---|
| Primary | 31 | Activate secondary control |
| Secondary | 0 | Virtualize APIC accesses |

Even with the VM-execution controls set according to Table 45, the Intel reference manuals still describe scenarios where access to the APIC physical addresses will not cause a VM-exit. Intel documents [Int12] cite the following cases where a VM-exit may not occur when accessing the physical pages for the local ASIC:

1. Large page translations - Intel defines linear access as (1) an access that results from a memory access using a linear address, and (2) when the access's physical address is the translation of that linear address. When using a linear translation, if the translation is using a 4-Kbyte page, the VM-exit will always occur. When using large page (2-MByte, 4-MByte, or 1-GByte) translations, the APIC-access VM-exit may or may not occur.
2. Improper Extended Page Table (EPT) invalidation - When EPT is enabled and the cache information is not properly invalidated from the EPT paging structures, the APIC-access VM-exit may be missed.
3. Improper TLB and Paging-Structure Caches - When Virtual Processor ID (VPID) is enabled and the TLB information is not properly invalidated using the INVVPID instruction, the APIC-access VM-exit may be missed.
4. Guest-Physical Accesses - When EPT is enabled and if the EPT PDPTE maps to a 2-MByte page or a 1-GByte page (bit 7 of the EPT PDPTE is 1) guest-physical access may not cause an APIC-access VM exit.

<u>The Startup InterProcessor Interrupt Attack:</u>
The VMX extensions provide a partial mitigation for the SIPI Attack. After the processor issues the VMXON instruction, when the system is running in VMX root mode (i.e., trusted hypervisor), the INIT and SIPI interrupts will be queued until the VMXOFF instruction is issued. The INIT and SIPI interrupts will be masked when the system is running in VMX non-root mode. This is only a partial mitigation due to the fact that the above race condition still exists. In

addition, the VMXOFF instruction is typically executed as part of the power down sequence, allowing this attack to execute. Wojtczuk and Rutkowska in [WRd] demonstrated this attack by placing shell code for playing a song. After the attack was executed and system shutdown initiated, the shell code song was played as the system shutdown.

Another mitigation strategy is to implement the Intel TXT-based trusted boot process using the SMX extensions. When the trusted hypervisor is started via the SENTER SMX instruction, the delivery of the INIT interrupt causes the platform to immediately shutdown if outside of VMX mode. In VMX mode, the delivery of the INIT interrupt will cause the platform to shutdown immediately once the VMXOFF instruction is executed. This mitigation strategy should provide proper protection when the TXT-based trusted boot process is not circumvented.

The same researchers who discovered the SIPI Attack, also demonstrated other attacks [WRa, WRT, WRb] to bypass the Intel TXT-based trusted boot process and compromise the SMM code on the platform. One attack [WRb] required Intel to: 1) Update the SINIT modules in the TXT-based trusted boot process to fix the buffer overflow for the affected processors, 2) Release updated processor microcode to prevent the loading and execution of the buggy SINIT modules, and 3) Add a preventative measure to permit blacklisting of buggy SINIT modules in the future.

These researchers believe the proper fix for TXT-based trusted boot process is the SMM Transfer Monitor (STM) to sandbox potentially malicious SMM code. Intel discusses this solution [Int12] as the Dual-Monitor Treatment of SMIs and SMM, but none of the major BIOS vendors or OEMs (including Intel) has implemented this solution [WRb].

System Management Mode:
In addition to the attacks used to bypass the TXT-based above, independent (but concurrent) research [DEG, WRc] identified additional attacks using the Memory Type Range Registers (MTRR). These registers are used to assign memory types (i.e., Uncacheable, Write Combining, Write-through, Write-protected, or Writeback). These attacks took advantage of the MTRR settings and changed the SMRAM range to writeback. The attacker then generates write access to the physical locations corresponding to the locations of the SMRAM (effectively poisoning the cache). Finally, the attacker triggers an SMI (which can be triggered simply by executing OUT to port 0xb2). When the processor enters SMM, the values from the cache may be used, allowing user code to be run with SMM privileges.

## 7.3 APPLYING THE SECURITY POLICY

The final step in the framework is to apply the Security Policy to the information flows identified in the previous step.

As described in Chapter 5, red networks are cleared to transmit classified and/or sensitive information in plaintext. Black networks are cleared to transmit encrypted classified and/or sensitive information. Figure 16 illustrates how two security gateways would be used to transmit classified and/or sensitive information between two red networks via a black network.

For this security policy, the Intel Nehalem Architecture would be configured to use TXT-trusted boot and the VMX extensions to run the guest operating systems. Each guest operating system would be responsible for a specific task (e.g., the encryption or the decryption of network traffic, providing guard services, Etc.).

The security policy used for this report:

1. Red network data is only permitted to be transmitted to a red network with the same classification. Separate cryptographic keys shall be used to ensure the separation of data at different classification levels.
2. All encrypted data must pass through the guard before transmission on the black network.

## 7.4 CONCLUSION

Based on the partial analysis of the boot sequence and interprocessor interrupts provided above, we cannot recommend the Intel Nehalem multicore architecture as a generic MILS multicore architecture for the following reasons:

- Thirty-five years of backwards compatibly adds a significant complexity and risk to the multicore architecture for high-assurance systems.
- The hypervisor privilege level (ring -1) is not native to the processor core, but bolted on using the VMX processor extensions.
- The required safeguard complexity makes hypervisor development expensive and error prone.
- The VMX processor extensions do not address potential timing channels.
- The SMM (ring -2) provides a higher privilege level than the trusted hypervisor.
- The processor cores permit a level of reconfiguration via Intel proprietary microcode updates[9].
- TXT-trusted boot does not provide protection from malicious code running in the SMM. The STM is designed to provide proper sandboxing for the SMM during the TXT-trusted boot, but no major OEMs or BIOS vendors (including Intel) have implemented it.
- The SMM is subject to cache poisoning if the MTRR registers are improperly configured.

[9] This means the same processor core may behave differently depending upon the version of the microcode that is loaded at boot time.

# 8 CONCLUSIONS AND FUTURE WORK

## 8.1 CONCLUSIONS

None of the multicore architectures analyzed in this research lend themselves to generic high-assurance MILS systems.

The CBEA in the Sony PlayStation 3 was analyzed in Chapter 5. This multicore architecture combined the PowerPC Processor Element (PPE) with eight Synergistic Processor Elements (SPE)s. The SPE processor cores had its own local memory store independent from the main storage domain. Initially, this memory isolation appeared to solve many of the memory and cache concerns present in multicore architectures using a shared main storage domain. Unfortunately, the CBEA is very reliant on the PPE to manage and control each of the SPE processor cores. This dependency requires that the PPE (the most complex component) be trusted and run a full Multi-Level Secure (MLS) environment to maintain separation between the SPE processor cores. This requirement eliminated the CBEA as a generic multicore architecture for high-assurance MILS systems.

A paper analysis on the Freescale P4080 multicore architecture was performed in Chapter 6. This multicore architecture initially showed good promise as the processor cores and CoreNet Coherency Fabric was analyzed. This architecture supports Asymmetric Multi-Processing (AMP) and provides protections via logical partitions to help ensure isolation between the abstract polyhedrons. This multicore architecture is also power sensitive, so the architecture supports the disabling of most components on the chip. This ability to disable unused components is powerful when mapping the security policy to the information flows and safeguards. Unfortunately, the area the Freescale P4080 multicore architecture suffered was in Data Path Acceleration Architecture (DPAA) and the peripheral bus controllers. Due to coupling between the components and peripherals, the abstract polyhedrons needed to expand to include multiple components. The grouping of the components reduced the usefulness of the Freescale P4080 multicore architecture as a generic high-assurance MILS system.

A brief paper analysis and a literature search for known vulnerabilities on the Intel Nehalem architecture was performed in Chapter 7. Thirty-five years of innovation and backwards compatibility has introduced a significant amount of complexity into this multicore architecture. For example, the multicore architecture does not support a native hypervisor privilege level, instead a set of processor extensions are used to introduce a new pseudo-privilege level (ring -1). This added complexity also shows in the amount of effort that is required to implement a safeguard to prevent the guest operating system from writing to the local Advanced Programmable Interrupt Controller (APIC). Additionally, this multicore architecture also implements a System Management Mode that runs at a higher privilege level than the hypervisor. Finally, the literature search identified several vulnerabilities which were able to run shell code on the processor when the VMX extensions were turned off, attack the SMM code, and then bypass the TXT-trusted boot loader. The complexity of this multicore architecture eliminated it from consideration as a generic high-assurance MILS system.

## 8.2 FUTURE WORK

The framework presented in this report provided the groundwork for simplifying and streamlining the analysis of multicore architectures for use in high-assurance MILS systems. Below are several possible improvements for extending the framework:

- **Complete System** - The analysis presented in this report groups the peripherals on an external bus as a cloud all running at the same clearance level. This simplified and made the analysis apply to general purpose use cases. Potential improvements to this framework would be a focus on these interfaces and external peripherals to see if the single clearance level requirement could be relaxed.
- **Shared Components** - This framework focused on major hardware components. The granularity may be improved in some of the more complex components (e.g., processor cores). For example, Simultaneous Multi-Threading (SMT) allows virtual processors to share the components of a single physical processor core. At a minimum, the polyhedron abstract concept presented in this framework would need to be modified to address the sharing of physical resources by the virtual processors.
- **Additional Security Policies** - This report focused on the identification of information flows and safeguards at the hardware and trusted hypervisor level. Not much time was spent attempting to map these information flows and safeguards to a wide variety of security policies. It was also interesting since we could not recommend any of the multicore architectures for the security policy we chose. With a different security policy, the Freescale P4080 multicore architecture may provide a suitable solution.

## 8.3 CONTRIBUTIONS

The initial evaluation of the CBEA multicore architecture for use in a MILS-compliant architecture sparked the idea for this research topic. The initial analysis used a relatively ad-hoc method, but an informed approach to multicore information flow analysis was used that was based on methods used for single-core information flow analysis. Before this framework, the analysis of the hardware and security policies was combined into a single analysis. Each analysis was focused through the lenses of a particular security policy. This required a complete reanalysis every time the same multicore architecture was evaluated for a different security policy. This effort was inefficient and so I started researching methods for separating the analysis of the multicore architectures from the security policies.

The first step of the framework is to identify all the hardware components to be evaluated. This step is very simple, but important. Without this step, it was not always clear if all the components had been evaluated. This list also provided a communication and status method that could be shared among multiple evaluators. Once the component was identified, the component was enclosed in an abstract polyhedron. The abstract polyhedron is also a major contribution to the evaluation of multicore architectures.

The second step is to identify the information flows and safeguards. The second step starts to illustrate the power of the abstract polyhedron. The abstract polyhedron simplifies and focuses the evaluation effort by only modeling the actions that breach the abstract polyhedron or alter the external visible state of the component (i.e., polyhedron color). The information flows represent the information flow capabilities of the hardware. Safeguards represent the hardware capabilities to reduce or restrict the hardware information flow capabilities. The completed model from this step is a contribution since it represents the important information flows (i.e., those information flows which can be used as overt or covert communication channels) and hides the non-important information flows (i.e., those information flows which are internal and not externally visible). Safeguards are added to this model to identify important information flows that can be reduced or restricted.

The final contribution presented in this project is the evaluation of the CBEA, Freescale P4080, and the Intel Nehalem multicore architecture for use in generic high-assurance MILS system.

# PART III

# SECURITY EVALUATION OF VIRTUALIZATION TECHNOLOGIES IN MULTICORE SYSTEMS

# 9 INTRODUCTION

This part of the report presents security evaluation of virtualization technologies in multicore systems. Multicore processors are becoming ubiquitous in the enterprises because they can be utilized to get higher performance by splitting system tasks into subtasks and distributing them across the multicore chipset. As one key component of multicore architectural features, virtualization technology has spurred a number of exciting development with virtual machines, especially in the area of hardware-assisted extensions. Intel and AMD have both released hardware virtualization support in their respective current mainstream processors, such as Intel Core i7 and AMD Phenom. However, while providing services in a virtual machine gains benefits, this has also led to explosive growth of security concerns in virtual machine systems.

To evaluate security of virtualization technologies in multicore architectures, this project first examined hardware features for virtualization technology in multicore processors, such as hardware-assisted Intel Virtualization Technology (Intel VT-x) and AMD Virtualization Technology (AMD-V) as well as hardware features in Cell Broadband Engine (CBEA) processors. Furthermore, by taking advantage of hardware virtualization support, we developed a lightweight hardware-assisted virtual machine monitor prototype, called IAVMM (for information assurance virtual machine monitor), which is purpose-built for the Intel 64 architecture and security analysis of multicore systems. Although hardware manufacturers are deploying multicore technologies, the science behind understanding the security of these systems is still lacking. In order to improve our understanding of security in multicore systems, we present and examine a layered framework for secure multicore architectures, and then introduce a layered assurance scheme for such architectures and illustrate how to formalize the framework based on 3 layers (from hardware layer, through hypervisor layer, up to user layer).

The work presented in this part of the report leads to a better understanding of virtualization technologies and corresponding security concerns in multicore architectures, which will benefit architects and researchers by providing security evaluation of hardware features and a lightweight virtual machine monitor. Moreover, a layered assurance scheme is provided to assist in the evaluation of security for multicore architecture design.

## 9.1 MOTIVATION

Multicore processors are becoming ubiquitous in the enterprise since they can be utilized to get higher performance by splitting system tasks into subtasks and distributing them across the multicore chipset. Therefore, vendors, such as Intel, AMD and IBM, are currently shipping processors with 4 or 8 cores, and are working on developing chipsets with more cores. As a crucial technology in multicore architectures, virtualization technology, which virtualizes the underlying physical resources that can be shared among multiple guest OSs, attracts more and more attention. For example, the National Security Agency (NSA) used to use separate physical machines to access networks with data at different levels of classification, which is of course expensive and unwieldy. The virtualization technology prompted NSA's move to their NetTop architecture in the late 90's, which uses virtual machines for isolation on the same physical host [MS00]. The work presented in this report focuses on such technology, especially hardware-assisted virtualization technology. With this work we are able to get a better understanding of virtualization technology and potential security concerns in multicore systems.

Virtualization has profoundly changed the information technology (IT) industry in a wide variety of areas such as operating systems, network, applications and storage. With virtualization technology, companies can gain scalability, since they can upgrade their virtual machines without necessarily upgrading the physical machines; companies can gain security, since they have a more controlled management environment and easier environment for backup and restore; companies can gain high availability, since they can share huge computational resources by cloud computing technology; companies gain convenience, since they can try or test some solutions without the necessity of using real environments. Along with the development of virtualization technology over the past decade, there has been steady activity and progress associated with multicore architectures, especially after the integration of hardware-assisted virtualization technology. The widespread use of hardware-assisted virtualization technology has spurred new research related with virtual machines (VMs). The addition of hardware-assisted virtualization technology to the x86 remedies problems found with virtualization of traditional x86 architectures [RI00], and supports a much simpler hypervisor implementation. Intel and AMD have both incorporated explicit hardware support for virtualization in their CPU chips since 2005 in order to provide efficient virtualization. Although people are already using virtualization hardware, there is still a need for an understanding of the science of these technologies. Therefore, we examine these current hardware-assisted virtualization technologies and corresponding hardware features in multicore architectures.

In addition, the virtualization technology development has also led to explosive growth in research efforts into the reliability of virtual machine systems (VMS). The concept behind virtualization technology is to virtualize underlying physical resources that can be shared among multiple guest OSs. The isolation properties of this technology have become attractive from a security perspective, as has the ability to inspect the details of a VM's execution or the ability to tweak the isolation boundaries [BLRS08]. For example, in a forensic team research, it is possible to clone a potentially compromised host into a VM and make further investigation without the need of the physical machine. The investigation team can also take advantage of snapshots to return to a previous state. There are other benefits of using virtualization, such as isolation for IDS or honeypots. However, while providing services in a virtual machine offers a comparatively harder target for attackers and thus gains benefits, it also introduces some new tricky security challenges (e.g. VM-escape[10], VM-Based Rootkits) [CN01, FO06, Rut08]. According to the National Vulnerability Database [NVD], there were 25 security vulnerabilities identified in VMware ESX [VMwa] in the last 3 years. Likewise, virtual machine technology is rapidly gaining acceptance as a fundamental building block in enterprise data centers. Since availability, integrity, and reliability are all critical challenges for these data centers, they are now using virtualization in an attempt to maximize the usage of their hardware resources by running on a single physical server instead of multiple servers. Therefore, it becomes increasingly important to investigate those critical components and verify secure system execution with virtualization technology, especially the popular hardware-assisted virtualization technologies in Intel and AMD architectures.

[10] VM-escape is the ability to bypass the virtual machine monitor and get access to the underlying system.

When developing secure computing systems, especially for high assurance and multilevel secure environments, it is important to understand the security capabilities of the underlying hardware as well as the relative merits of different VMM features, functionalities and configurations. In order to address this efficiently, we were interested in a small VMM kernel that can be used as a platform for experimentation relevant to security features of multicore processors with hardware virtualization support.

The design of a secure system requires architects to develop a system architecture that supports implementation of various security policies. Enforcing security policies at the architecture level is attractive because it allows security concerns to be recognized early and can be given sufficient attention in the design stages. A security policy, determined by regulation and doctrine, defines "secure" for a system.

## 9.2 RESEARCH CHALLENGES

Hardware-assisted virtualization technology has been presented to industry as an efficient solution to improve virtualization. There are still many features to be examined from a security perspective when utilizing such hardware extensions in Intel or AMD architecture. The very first question we face is: can hardware-assisted virtualization technology provide more security than software-assisted virtualization technology? To answer this, we need to evaluate hardware-assisted virtualization technology, and determine the resources that are virtualized in these hardware extensions, and make a comparison of security protection features provided by software and hardware virtualization.

The next issue that arises is how to evaluate whether these hardware extensions can provide a more secure virtual machine system. To address this issue, a Virtual Machine Monitor (VMM) is designed and implemented for security analysis of the underlying hardware architectural features. We are interested in a small VMM that allows us to build experimentation based on that platform. Compared with commodity hypervisors, which are always of a huge size, i.e., Xen 3.4 consisting of 200 KLOC (thousand lines of source code) and VMware consisting of 230 KLOC, this VMM should be lightweight and convenient for extensible development. Furthermore, to avoid the possible effects of existing host operating systems, this VMM should run directly on the top of "bare metal", which means that no system API support will be provided to develop the VMM. Last but not least, assuming that the hypervisor is supposed to be secure and impenetrable, we still need to evaluate the security exploits possibly existing in the virtual machine systems. Specifically, this challenge can be split into the following problems:

1. Whether a program running on a guest VM can interfere with other guest VMs or VMMs unless an explicit communication has been established?
2. Whether there exists a covert channel that makes keeping secure data on one VM safe from other VMs very difficult?
3. Whether there exist Denial of Service (DoS) attacks on multicore systems, such as IBM CBEA processors and Intel Core i7 series processors?

All of these questions have two avenues of explanation. First, can the guest VM utilize hardware resources of the architecture to affect another VM? Second, can the guest VM cause a

VM escape? In this project, we are primarily interested in the first avenue. Although the second is very important, the answer depends primarily upon the implementation of the VMM. We will explore the hardware features that can be used by the VMM to protect it.

In addition, the design of a secure system requires architects to develop a system architecture that supports implementation of various security policies. Enforcing security policies at the architecture level is attractive because it allows security concerns to be recognized early and can be given sufficient attention in the design stages. Therefore, our research efforts are also taken to enforce security policy for multicore architectures. The term "security policy for multicore architectures" covers security requirements that protect multicore systems from any unauthorized behaviors. To develop a secure multicore system, it is necessary to enforce a security policy framework as simple and as strong as possible to provide general guidance for secure systems on multicore architectures. Due to a wide variety of malicious behaviors, assurance would most likely have to specify exactly what each element of software is intended to perform, and to provide evidence that it does it correctly. Similarly, we analyze a multicore architecture by examining each component from hardware level, through hypervisor level up to user level. Briefly, mapping a higher-level (user level) security policy into the supporting security policies in lower level (e.g. infrastructure hardware) is a complex process which is not well verified and supported by current security engineering techniques, and this research provides a practical example of how to perform this verification.

## 9.3 CONTRIBUTIONS

The goal of this part of our project is to offer a compelling approach to enhancing secure virtual machine systems with virtualization technology by evaluating hardware-assisted virtualization technology in current multicore systems, building an experimental VMM platform residing on the multicore architecture and formalizing a security policy framework for multicore architectures. To be specific, the contributions of this project are as follows:

- Examine and contrast current hardware-assisted virtualization technologies in multicore architectures, including Intel Virtualization Technology (Intel VT-x) and AMD Virtualization (AMD-V). Moreover, we evaluate the hardware features for virtualization technology in multicore processors, especially Intel Core i7-860. In addition, we also evaluate security concerns of the CBEA processor. There are only a few books and documents dedicated to these subjects.
- Design and develop our lightweight virtual machine monitor, IAVMM, for security analysis of the underlying Intel 64 architectural features. The IAVMM is a Type I VMM and runs directly on the processor (i.e., bare metal). It's a very lightweight VMM, only consisting of seven thousand lines of code. IAVMM doesn't provide the full functionality of commodity hypervisors, but enough for research experimentation, and it is extensible if more features need to be implemented in this prototype. Based on this platform, some potential vulnerabilities are pointed out, involving registers, instructions, and shared memory issues.
- Propose an approach for specifying and verifying a layered assurance scheme for multicore architectures. Multicore architectures are analyzed by examining a layered security policy framework, from hardware level, through hypervisor level up to user level. After identifying and examining multicore hardware architectural features and

enforcing necessary security policies in VMM level, we decompose the policies into components that can be mapped into hardware level, then verify that VMM- and Hardware- level security policy satisfies user-level security requirements.

In summary, the work presented in this report leads to two practical security engineering techniques, including a lightweight virtual machine monitor for security analysis of Intel 64 architectural features and a layered security policy framework, which will benefit system architects by providing design guidance and reducing overall design efforts in developing secure multicore systems.

## 9.4 PART III OVERVIEW

Chapter 10 provides a brief review of the relevant background, including multicore processors and corresponding hardware-assisted virtualization technology, and then describes previous and related work about hypervisors with security concerns and security architecture modeling.

Chapter 11 describes and compares the hardware-assisted virtualization technology, including Intel VT-x and AMD-V. In addition, CBEA processors are also examined from a security perspective.

The development and implementation of our lightweight VMM, IAVMM, are described in Chapter 12. IAVMM is provided for security analysis of the underlying Intel 64 architectural features. It's convenient for us to build research experiments based on this prototype. The validation of this IAVMM and some security concerns of multicore architectures are presented in Chapter 13.

Chapter 14 models a hierarchy of a layered security policy framework in terms of multiple secure architectures that are related by formal mappings, and provides an exemplary 3-layer version of the model, from hardware level, through hypervisor level up to user level.

Chapter 15 introduces a formal model of virtual machine systems and highlights out a security requirement for multicore architectures, followed by some examples of this formal model.

Finally, Chapter 16 summarizes the work of Part II of this report and introduces a number of directions for future work in the areas of security challenges in multicore architectures.

# 10 BACKGROUND

This chapter presents background research in the area of virtualization technology, especially in multicore systems. We start with a summary of terminology, followed by an overview of current multicore processor architectures, then a discussion of virtualization technology for different components of a virtual machine monitor. The related work about security challenges in multicore systems is also presented.

## 10.1 MULTICORE ARCHITECTURE TERMINOLOGY

The following are the basic terms and concepts used in virtualization and multicore architectures that are relevant to this project.

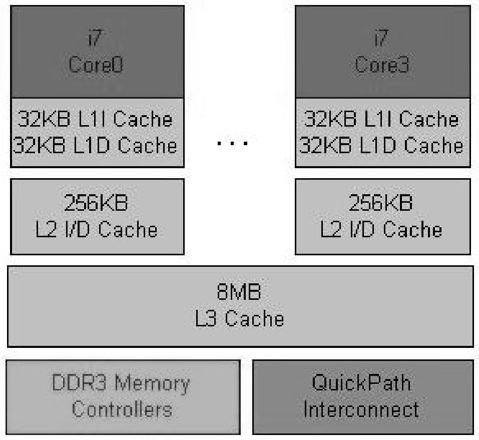
- **Multicore Processor:** A multicore processor is a physical package that contains more than one processor core.
- **Physical Package:** The physical package is a microprocessor. Each physical package is plugged into a physical socket on a main board, and may contain one or more processor cores.
- **Processor Core:** A processor core is the circuitry that provides the ability to decode and execute instructions. A processor core may contain one or more logical processors.
- **Logical Processor:** A logical processor is the basic unit of processor hardware that allows the software executive in the operating system to dispatch a task or execute a thread context. Each logical processor can execute only one thread context at a time.
- **Hyper-Threading:** Hyper-threading is a feature where each processor core provides the functionality of more than one logical processor.
- **Advanced Programmable Interrupt Controller (APIC):** An APIC is either an I/O APIC or a local APIC. It is attached to the APIC bus, which is a special non-architectural bus on which the APICs in the system send messages. The I/O APIC is a specially designed for receiving and distributing interrupts from external devices. The Local APIC is built into the processor and is responsible for dispatching interrupts sent over the APIC bus to its processor core, and sending interrupts to other processors over the APIC bus.
- **Virtual Address:** A virtual address is the memory address used by a process when specifying the address of an operand or an instruction. This address is translated by the system into a physical address. Sometimes it is referred to as a logical address. We call this address the guest virtual address within a guest VM.
- **Physical Address:** A physical address is used by hardware to address memory cells included in memory chips in order to enable the data bus to access a particular storage cell of main memory.
- **Logical-Partitioned System:** In a logical-partitioned system, partition-management software (a hypervisor) creates the partitions by assigning portions of the total system resources to each partition, each capable of running its own operating system and user environment such that a program executing in one partition cannot interfere with any program executing in a different partition.

- **Virtual Machine Monitor (VMM):** A VMM, also known as a hypervisor, is software that arbitrates access to the underlying physical resources which can be shared among multiple guest operating systems. A VMM acts as a host and has full control of the processor(s) and other platform hardware.
- **Virtual Machine (VM):** A VM is a software implementation of an isolated machine that executes programs like a real machine.
- **Guest Software:** Each Virtual Machine (VM) is a guest software environment that supports a system consisting of operating system and application software.

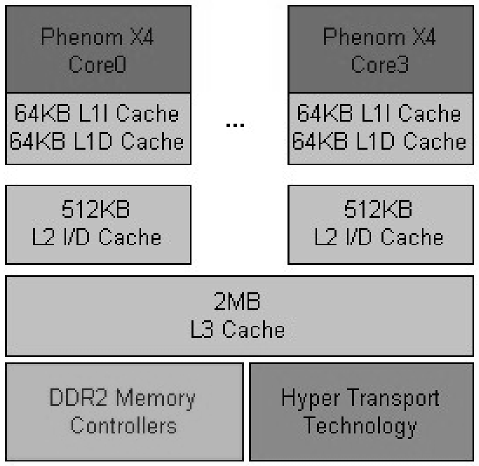
## 10.2 OVERVIEW OF MULTICORE PROCESSORS ARCHITECTURES

This section provides a review of the Intel Nehalem, AMD Phenom and IBM CBEA architectures. Both Nehalem and Phenom architectures support hardware-assisted virtualization technology, while CBEA supports logical partitioning technology.

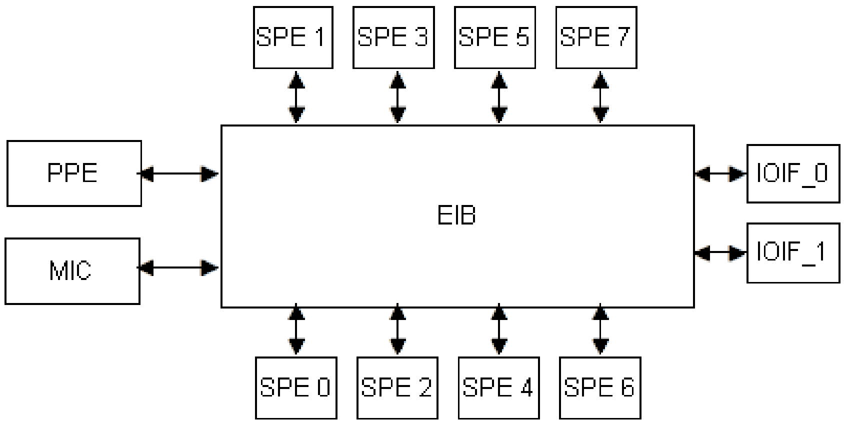
As Intel's latest flagship processor, the Nehalem processor, such as Intel Core i7-860, is a 64-bit processor fully integrated with four cores, and an inclusive and shared L3 cache as shown in Figure 30(a). From a performance perspective, the inclusive L3 cache is the ideal configuration since it keeps most cache coherency transactions on-die. However, it comes with potential security challenges as well, since the L3 cache is shared by all four processor cores. The Nehalem architecture provides Hyper-Threading technology for parallel, multi-threaded execution by reducing computational latency and making optimal use of every cycle. To achieve performance advantages with the integrated quad-core, Nehalem also replaces the front-side bus with an integrated memory controller and on-die dedicated interprocessor QuickPath Interconnect (QPI), which is "a packet-based, high bandwidth, and low latency point-to-point interconnect" [Kan08]. AMD built a processor (AMD Phenom X4 quad-core processor) after the release of Nehalem and claimed at that time that it was the industry's first true quad-core x86 processor since the cores can communicate on-die rather than on package for better performance. In Figure 30(b) we sketch the Phenom quad-core processor: each core has its own L1- and L2-caches, while all cores share a common on-chip L3-cache. Similar to Intel core i7, this design in multicore CPUs allows all cores to process data at a rate close to clock rate and to reduce memory latency. Meanwhile, Phenom integrates the memory controller on-die and uses multiple memory banks to improve memory throughput. If the caches and the memory are concurrently accessed by all cores, contention for their utilization may increase the latency of memory operations and degrade performance. Phenom replaces the front-side bus, which is currently different for every type of machine, with an open specification, HyperTransport controller. As with QPI, HyperTransport is a packet-oriented, power-managed, multi-link high speed interconnect.



(a) Intel Nehalem architecture on a single die



(b) AMD Phenom X4 architecture on a single die



(c) CellBE architecture

**Figure 30: Overview of multicore processor architectures**

The CBEA processor is an implementation conforming to the Cell Broadband Engine Architecture (CBEA), which extends the 64-bit PowerPC architecture. Both the CBEA and the CBEA processor are the results of a collaboration among SONY, Toshiba and IBM, known as STI, formally begun in early 2001 [IBM07b]. In a CBEA processor (Figure 30(c)) we have a core processor, called PowerPC Processor Element (PPE) which controls tasks and eight Synergistic Processor Elements (SPEs) for data-intensive processing. The SPE consists of the synergistic processor unit (SPU) and the memory flow control (MFC) responsible for the data movements and synchronization, as well for the interface with the high-performance Element Interconnect Bus (EIB). The PPE has a L1 instruction cache and a L1 data cache as well as a unified L2 instruction and data cache, whereas the SPU only has its unified 256 KB Local Store (LS). The hypervisor in the CBEA processor is typically implemented in a small executive that is packaged as firmware. It creates the partitions by assigning portions of the total system resources (the PPE, the SPEs, the I/O controller, and the EIB) to each partition as in a logical-partitioned system. Of course it harvests those resources when that partition is deleted.

## 10.3 VIRTUAL MACHINE MONITOR

A VMM, also known as a hypervisor, is software for a computer system that arbitrates access to the underlying physical resources which can be shared among multiple guest operating systems (OSs). These guest OSs are referred to as virtual machines (VMs). Classic microprocessors implement levels or rings of execution, with at least a supervisor and user mode. Applications run in user mode, while the operating system runs in supervisor mode. To provide virtualization and execution of concurrent operating systems in separate VMs, there is a need to support these execution levels in each VM and still protect the VMM. This can be done through the use of software and/or hardware. Popek and Goldberg's 1974 paper [PG74] describes the characteristics for classical virtualization through three essential requirements:

- *Equivalence or Fidelity.* A program running on a virtual machine must exhibit identical behavior as if it was running on an unvirtualized machine;
- *Efficiency or Performance.* The majority of operations must be performed on real hardware resources without the intervention of the virtualization layer;
- *Resource Control or Safety.* The virtualization layer must completely control real system resources. No program running on a virtual machine can access any resource not explicitly allocated to it by the hypervisor.

The definition of a VMM does not specify how the VMM gains control of the machine to interpret instructions that cannot be directly executed on the processor. As a result, there are two different types of VMMs that can create a virtual machine environment. Type 1, also called native or bare-metal, runs directly on top of the system's hardware on real ring 0 to control the resource and to monitor guest OSs. A guest operating system thus runs on another level above the VMM, allowing for true isolation of each virtual machine. For instance, VMware ESX Server [VMwa] and Xen [Xena] are Type 1 hypervisors. Our tiny IAVMM is also a Type 1 VMM. Type 2, also called hosted, runs within a conventional operating system environment, usually in ring 3; that is, the VMM operates as an application on top of an existing operating system and provides only virtualization support services. This type of VMM has a lower performance than the other type because factors such as calls to the hardware must traverse many layers before the operations are returned to the guest operating system. Both VMware workstation [VMwb] and Oracle VM VirtualBox [Vir] are Type 2 VMMs.

The term *classically virtualizable* is used to describe an architecture that can be virtualized purely with the trap-and-emulate approach, a prevalent VMM implementation method proposed by Popek and Goldberg [PG74]. In this approach, when a guest OS is attempting to execute an instruction for which it needs the supervisor privilege level, the CPU is able to "trap" such attempts, and allows the VMM to "emulate" the effect that is desired by that guest OS. However, applying this virtualization technology on traditional x86 architecture becomes complicated due to an inherent lack of support, i.e. address-space compression11, ring aliasing12, ring

[11] OSs expect to have access to the processor's full virtual address space, some portion of which is reserved by the VMM. The VMM must prevent guest access to those portions to protect its integrity.

[12] Ring aliasing refers to problems that arise when software is run at a privilege level other than the privilege level for which it was written.

compression13 and so on [AA00, NSL+06, RI00]. Fortunately, recent architectural modifications with hardware extensions for virtualization (Intel VT-x and AMD-V) make the x86 architecture classically virtualizable. The following subsections discuss the evolution of virtualization for CPU, memory and I/O components.

### 10.3.1 CPU Virtualization

In this section we discuss CPU virtualization. We specifically focus on two primary features: privilege levels and CPU instructions.

In the x86 family, the processor's protection mechanism recognizes 4 privilege levels[14], with value 0 denoting highest privilege level and value 3 for the lowest one. Levels 1 and 2 are not typically used in most of operating systems. The purpose of this division is to allow a microprocessor to determine whether a privileged instruction can execute without fault or exception, thereby increasing system stability and reliability. Although there are four available levels, guest VMs traditionally only run at level 3 with restricted access, whereas the VMM operates within level 0 which enables total access to platform resources, like CPU and memory. However, a guest OS in a VM acts as if it was a real OS and will attempt to execute some privileged instructions, and will try to limit processes to higher privilege levels. This results in *ring compression*, where a guest OS and its applications must both run at level 3, unprotected by hardware from user applications. Therefore, this division of privilege levels conflicts with the virtual machine systems. Intel VT-x addresses this issue by providing two forms of CPU operation, VMX root operation and VMX non-root operation, both of which support all four privilege levels, allowing a guest OS to run at its intended privilege level (privilege level 0).

Another challenge for CPU virtualization relates to CPU instructions. Besides those legacy instructions[15], Popek and Goldberg [PG74] introduce a classification of the instructions of an instruction set architecture (ISA) into two different categories:

- *Privileged* instructions that trap only if the processor is in user mode and have no trap if it is in supervisor mode, such as Intel instructions *VMCALL*[16], *CPUID*[17] and the instructions for accessing I/O device ;
- *Sensitive* instructions that try to change the configuration of actual resources in the hardware platform, such as Intel instructions *SIDT*[18],*SLDT*[19].

[13] The meaning of "ring compression" used here will be discussed in detail in the next section.

[14] Sometimes these privilege levels are designated as privilege rings.

[15] Legacy instructions, as executed by the virtual machine, obey the equivalence property [PG74].

[16] The VMCALL instruction allows guest software to make a call for service into an underlying VM monitor.

[17] The CPUID instruction returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX.

[18] The SIDT instruction stores the content of the interrupt descriptor table register (IDTR) in the destination operand.

[19] The SLDT instruction stores the content of the local descriptor table register (LDTR) in the destination operand.

According to Popek and Goldberg, when a VM attempts to execute a sensitive instruction, it is guaranteed to trap automatically and signal the VMM. In the x86 architecture, there are 17 sensitive but unprivileged instructions identified by Robin and Irvine [RI00], which don't meet this requirement. In other words, the control is not passed back to the VMM when these sensitive instructions are invoked, and thus the VMM cannot emulate the expected behavior. For example, the POPF instruction, one of sensitive and unprivileged instructions, pops a word from the top of the stack and stores the value in the lower 16 bits of the EFLAGS register, hence allowing values in the EFLAGS[20] register to be changed. Therefore, it is unsurprising that the traditional x86 architecture fails to be classically virtualized.

In order to overcome these aforementioned limitations, some new techniques [VMw07] have been proposed to implement CPU virtualization on x86 architectures instead of the trap-and-emulate mechanism. These techniques include binary translation, paravirtualization and hardware-assisted virtualization.

Binary translation is the emulation of one instruction set by another through translation of code. Hence, an additional binary translator is needed to handle the entire x86 instruction set for this mechanism. Moreover, the translator must provide the ability to intercept precisely the virtualization sensitive instructions without requiring trap semantics. For instance, we can execute bytecode on a Java Virtual Machine. This binary translation mechanism only works when recompiling source code is not desirable or feasible.

Paravirtualization, as a different approach to overcome the virtualization issues, refers to a mechanism to efficiently improve the communication between guest OS and the VMM. The approach requires replacing non-virtualizable instructions on the guest OS with hypercalls that communicate directly with the virtualization layer, and providing hypercall interfaces for other critical kernel operations such as memory management. In other words, the running guest OS must be modified in order to operate in the virtual environment. Therefore, its compatibility and portability are poor. Take the Xen system with versions before 3.0 as an instance, the guest OSs needed to be modified to run on the Xen VMM successfully.

Hardware-assisted virtualization (i.e. Intel VT-x and AMD-v), is designed to provide new CPU execution mode features that allow the VMM to run in a new supervisor (host) mode. Moreover, privileged and sensitive calls can be set to automatically trap to the VMM with the support of hardware extensions, which means that we can eliminate the need of either binary translation or paravirtualization on the x86 architecture. We will discuss more details of this technology in Chapter 11.

### 10.3.2 Memory Virtualization

The memory management facilities of the IA-32 architecture are divided into two parts: segmentation and paging. The CPU control unit transforms a virtual address (logical address) into a linear address by means of a hardware circuit called a segmentation unit, and a second hardware circuit called a paging unit transforms the linear address into a physical address.

[20] The EFLAGS register contains the system flags and fields that control I/O, maskable hardware interrupts, debugging, task switching and the virtual-8086 mode. Only privileged code should be allowed to modify these bits.

Segmentation provides a mechanism for dividing the processor's addressable memory space (called the linear address space) into smaller protected address spaces called segments. A logical address, consisting of a segment selector and an offset, is provided to locate a byte in a particular segment. The offset part of the logical address plus the base address for the segment thus forms a linear address. If paging is not used, the linear address space is mapped directly into the physical address space. Otherwise, page tables are needed for the translation.

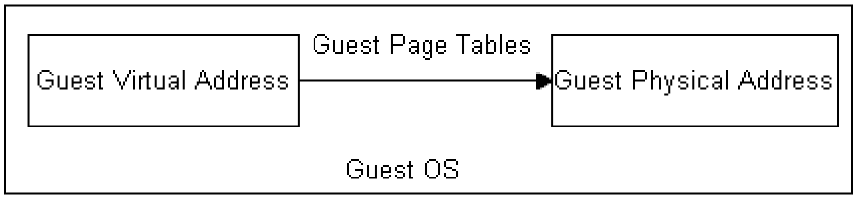
VMMs must control physical memory to ensure VM isolation and to remap guest physical addresses into host physical address space for virtualization. Memory virtualization allows the VMM to enforce control of physical memory and yet support guest OSs' expectation to manage memory address translation. In a VM-based system, we have at least two separate primary OSs (host OS and guest OSs) with each supporting their own translations. Therefore, virtualization for memory systems brings more challenges if we wish to allow both levels of OSs to control isolation within their domains of influence. In VM systems, host physical addresses are the only addresses that are used to access real memory cells. In a traditional microprocessor the Memory Management Unit (MMU) hardware support allows for quick translation of virtual/linear addresses to physical addresses by use of Translation Lookaside Buffers (TLB), which cache recently used page translations. When a TLB miss occurs, the hardware (or software in some systems) walks the page tables to obtain the new translation and updates the TLB. In a VM system, the guest OSs maintain a set of guest page tables that translate virtual addresses to guest physical addresses, and then the guest physical addresses will have to be brought out of the VM and be translated to host physical addresses. Before the advent of hardware support for MMU virtualization on chipsets for VM systems, this last step was implemented by the VMM through the use of shadow page tables (see Figure 31) during every TLB miss. However, under this shadow paging mechanism the VMM needs to intercept guest page table updates to keep the shadow page tables coherent with guest page tables, which wreaks havoc on the performance of this solution to memory virtualization. The reason is that the guest virtual address is commonly possessed by both shadow page tables and guest page tables. When the guest attempts to schedule a new process on the processor, it updates the processor's CR3[21] to establish the guest page tables corresponding to the new process. The VMM must intercept this operation and set the real CR3 value based on the corresponding shadow page tables for the new process. Therefore, frequent context switches within the guest could result in significant hypervisor overheads.

Rather than have the VMM manage the shadow page table's mapping in software, Intel's Extended Page Tables (EPT) and AMD's Nested Page Tables (NPT) solve this problem by adding a separate set of hardware-walked page tables which translate from guest physical addresses to host physical addresses that are used to access memory. To reduce frequent TLB flush[22], a CPU with hardware support caches the translation information in the TLB associated with a VM specific tag, such as Intel's Virtual Processor Identifier (VPID) tag or AMD's Address Space Identifier (ASID) tag. In Intel VT-x, VPIDs introduce to VMX operation a facility by

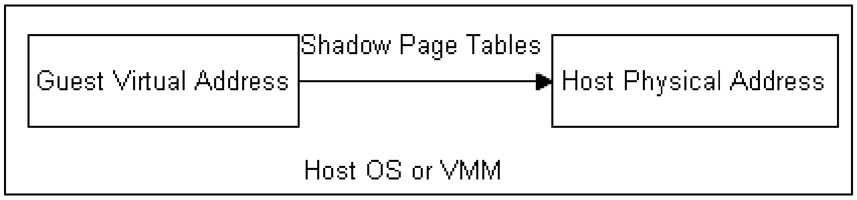
[21] In Intel processors, Control Register 3, CR3, contains the base memory address of the current top-level page table which stores the translation from linear to physical address.

[22] Normally, the entries in the x86 TLBs are not associated with any address space. Hence, every time there is a change in address space, such as a context switch, the entire TLB has to be flushed.

which a logical processor may cache information for multiple linear-address spaces. In this way, VMX transitions may retain cached information and the logical processor switches to a different linear-address space. More details will be provided in Chapter 11.



(a) Address Translation of Guest Page Tables in Guest OS



(b) Previous Address Translation Mechanisms in Virtual Machine System

**Figure 31: Page Translation Mechanisms**

### 10.3.3 I/O Virtualization

Before looking at how I/O devices are virtualized, it's important to know how conventional Peripheral Component Interconnect (PCI) [PCIa] and PCI Express (PCIe) [PCIb] work. Each PCI device is identified by a triple {bus, device, function}. A computer might have several PCI buses that could be linked or independent, but only one PCI device can use the bus at any given moment. It can't allow multiple devices to be active simultaneously on that PCI bus. Fortunately, PCIe, a new computer expansion card standard designed to replace the older PCI standard, is a point-to-point architecture rather than a shared parallel bus architecture. In PCIe, there are no direct connections between devices and each device is connected to a controller. Thus, this topology alleviates PCI's shared bus problem.

The VMM virtualizes the physical hardware and allows each virtual machine a set of customizable virtual devices. Most of this virtualized I/O requires software drivers that run on the host operating system to access the real hardware. If the VMM is a Type II hypervisor, it will use the device drivers already existing in the host OS, otherwise Type I hypervisors may need to provide their own device drivers for the hardware on the machine, as in the case of VMware ESX. Three approaches are provided to implement such a mechanism. The first approach to virtualizing I/O devices is emulation in software, similar to binary translation in CPU virtualization. With this approach, every time an I/O operation occurs, the VMM has to trap and emulate it, in turn causing additional data transfers and interrupts. As a consequence, if the device performs hundreds of megabytes of data transfer, the overhead is substantial and it degrades performance. The second is the paravirtualization approach for I/O devices, which is similar to paravirtualization for CPU as well. The hypervisor exposes relatively high-level APIs to guest OSs, and the guest OSs (or device drivers) need to be modified in order to apply these APIs for the corresponding operations.

In order to gain native performance, I/O virtualization allows virtual machines to talk directly to hardware. Two issues come along with this approach. The first one is if a device has already been assigned to one guest, it can't be reassigned to any others. To solve this problem, Xen applies a mechanism that requires a requesting guest OS to use virtualized drivers to send I/O requests to the I/O-assigned guest OS, and then the I/O-assigned guest OS communicates with the hardware instead of the requesting guest OS. The second big issue involves shared memory. The memory that I/O devices use for DMA transfer is all based on physical memory addresses, which don't correlate to the bounded virtual addresses visible to each guest. As a result, whenever the guest's driver directs the I/O device to perform DMA, it will fail because of the wrong memory addresses. To solve both of these problems, Intel and AMD introduced hardware extensions I/O virtualization technology Intel VT-d [Int08] and AMD-Vi [AMD09], respectively as a third approach to I/O virtualization. Since an I/O device can normally be assigned to exactly one guest virtual machine, PCIe has been extended in hardware extensions to solve this multiplexing problem so that I/O devices can offer multiple virtualized functions to different VMs [Bri]. Each triple {bus, device, virtual_function} can be assigned to a different VM, thereby allowing the I/O device to be shared. However, this requires device support and will only exist with newer devices. To fix the second issue, an IO memory management unit (IOMMU) is added in hardware extensions. The IOMMU not only provides the translation between guest physical addresses and host physical addresses automatically, but also provides some memory protection like restricting the physical address range a device can access to. More details will be provided in Chapter 11.

### 10.3.4 Interrupt Virtualization

The IA-32 architecture uses 8-bit vectors, of which 244 (20H-FFH) are available for external interrupts. Vectors are used to select the appropriate entry in the Interrupt Descriptor Table (IDT). Providing support for external interrupts, especially regarding interrupt masking, presents some specific challenge to VMM design. The IA-32 architecture uses the interrupt flag (IF) in the EFLAGS register to control interrupt masking. A VMM will likely manage external interrupts and deny a guest OS the ability to control interrupt masking by ensuring that guest attempts to control interrupt masking will fault in the context of ring de-privileging[23]. Such faulting could cause problems because some OSs may frequently mask and unmask interrupts, thereby affecting system performance significantly.

Even if it were possible to prevent guest modifications of interrupt masking without intercepting each attempt, challenges would remain when a VMM has a “virtual interrupt” to deliver to a guest. In Intel VT-x, a virtual interrupt is delivered only when the guest has unmasked interrupts. To deliver virtual interrupts in a timely way, a VMM should intercept

[23] Ring deprivileging is a technique that runs all guest software at a privilege level greater than 0.

some, but not all, attempts by a guest to modify interrupt masking. Doing so could significantly complicate the design of a VMM.

In summary, we have discussed the virtualization of important complex components of an entire system, such as virtualization of CPU, memory and I/O devices. As mentioned above, hardware-assisted virtualization of these components improves virtualization efficiency.

## 10.4 RELATED WORK

Virtualization technology for commodity processors has entered the hardware extensions era. Machines with hardware-assisted virtualization technology have become a prevalent platform for building virtual machine systems. In the meanwhile, a variety of research issues regarding security concerns are raised along with the development of this technology. In this section, we present some current prevalent VMMs and describe how they work, and then introduce the security challenges posed in multicore systems.

### 10.4.1 Previous Hypervisors

Xen, initially a paravirtualization open-source VMM, runs on bare-metal for the 32-bit x86 processor architecture and allows the user to run several guest OSs on a single host concurrently. Due to the addition of hardware virtualization extensions from Intel VT-x and AMD-v since Xen 3.0, Xen architecture can be split into three important and distinct domains: Dom0 (Domain Zero), DomU (Domain User) and HVM (Hardware-assisted virtual machine) domain. Dom0 is the first domain started by the Xen hypervisor on boot. It is a privileged service domain which is used to create and configure all of the remaining guest domains (DomU and HVM Domain). DomU, started by the xend command in Dom0, is an unprivileged domain with (by default) no access to the hardware. Xen-HVM has hardware-assisted device emulation to provide I/O virtualization to the virtual machines. In other words, contrary to common paravirtualization VMMs, current Xen allows unmodified guest operating systems in the HVM domain as well as paravirtualized guest OSs in Dom0 and DomU [BDF+03, DLM+06]. In addition, it also supports 64-bit platforms and 64-bit guests. Xen presents to each HVM guest a virtualized platform by the use of virtual modules, such as a virtual CPU module providing the abstraction of a processor, a virtual memory management unit (MMU) module presenting the abstraction of the hardware MMU, and a virtual I/O device in Dom0 providing the abstraction of a PC platform. Although Xen consists of a large amount of source code (approximately 230,000 lines of source code in Xen 3.4.1), it is useful as a building block towards secure VMMs. For example, Ether [DRSL08], a transparent malware analyzer residing completely outside of the target OS environment, makes use of Xen HVM and its support for Intel VT-x for malware analysis. Microsoft Hyper-V [Hyp] is also based on a Xen-like but not identical architecture.

VMware Workstation is a type II hypervisor, residing on top of a host OS and able to create virtual machines for a variety of guest OSs, such as Solaris x86, FreeBSD, Windows and Linux. In 2001, VMware Elastic Sky X (ESX) [VMwa, VMw07] was announced as the first server product with an approach different to that of the workstation version. VMware ESX doesn't require a host OS, but instead it has its own native hypervisor on a bare-metal system. Because the source code is not released to the public, we don't discuss it further in this report.

Microsoft Hyper-V is a type I VMM, which runs on top of bare metal. It's a part of Windows Server 2008 and it is a native 64-bit hypervisor that can run 32-bit and 64-bit VMs concurrently.

The main goal is to enable the possibility to run multiple guest OSs, called partitions, on a single server hardware system. There are two types of partitions: 1) parent partition: the controlling partition for the management of other child partitions, and managing and assigning the hardware devices with exception of processor scheduling and physical memory allocation since they are handled by the hypervisor; 2) child partition: this is where the guest OS will run, and is created by the parent partition. Partitions communicate with the hypervisor layer by using hypercalls, which can be considered APIs used by partitioned OS. As with VMware, Hyper-V is not an open-source hypervisor, so we don't discuss it further in this report.

At present, we are aware of another two systems which use AMD-v technology. Tiny Virtual Machine Monitor (TVMM) [Kan06] is a useful proof-of-concept lightweight VMM for understanding AMD SVM extensions. However, TVMM essentially only supports AMD64 architecture with virtualization technology and it only can boot its own sample operating system. Booting a real OS or a general purpose program requires many changes and proved to be a daunting task. Malware Analysis Virtual Machine Monitor (MAVMM) [NSJ$^+$09], an enhanced version of TVMM, is proposed as a yet-another lightweight hardware-supported virtualization platform that is purpose-built for malware analysis in AMD64 architecture. It can handle different paging modes, CPU operating modes and several types of guest images. Similar to TVMM, there are some deficiencies existing in MAVMM. First, it is a host VMM only in support of one single guest VM running at any given moment. Second, no input service is provided, which means that it is not easy for the system to receive commands or messages from external devices. Both of these VMMs can only support the AMD64 architecture, and currently only boot on the AMD SimNow simulator.

### 10.4.2 Security Challenges in Multicore Systems

As we have seen in the previous section, hardware virtualization technology is going mainstream. Security, as a very important component, has also become an import and timely research field. Our research efforts are dedicated to the security challenges posed in multicore systems and the corresponding defense systems proposed to mitigate these potential compromises.

A VMM segments physical resources into isolated entities and allows each guest OS to run independently, without affecting any other VM or the host OS. Unfortunately, current VMMs do not offer such perfect isolation. Hence, this isolation will be compromised if a VM-escape happens. Generally speaking, in VM-escape the program running in a guest VM is able to bypass the VMM layer and get access to the host OS[24], thus gaining root privileges and having full control over the computer. For example, the vulnerability for CVE-2007-4496 in VMware Workstation [VMWc], found by Rafal Wojtczuk in 2007, allows authenticated users with administrative privileges on a guest OS to corrupt memory and potentially execute arbitrary code on the host OS via unspecified vectors. The vulnerability for CVE-2007-4993 in Xen 3.0.3 [Xenb] found by Joris Van Rantwijk in 2007, allows the root user in a guest domain to trigger execution of arbitrary Python code in Dom0 by crafting a grub.conf file.

[24] In this case, we talk about the VM escape in Type II hypervisor systems.

Inspired by VM systems, a VM-based rootkit (VMBR), as a new type of malware software, is capable of installing a VMM underneath an existing operating system and hoisting the original target operating system, on-the-fly, into a VM that is then monitored and controlled. Corrupting an operating system in this way is particularly interesting from the attackers' point of view because it signifies corrupting all the software that run upon this host OS. VMBRs are more difficult to detect than traditional malware software since they gain full control of the system and thus their states cannot be directly accessed by security detection software (i.e., Intrusion Detection Systems (IDS)) residing in the target system. SubVirt [KCW+06] is such a VMBR. Luckily, defenders have been exploring techniques to detect the presence of a VMBR. A VMBR does leave signs of its presence that a determined IDS can observe, because a VMBR always tends to require a reboot in order to be installed before it can run, thereby having more of an impact on the system. Nevertheless, a new variant VMBR Blue Pill [RT07] was proposed by The Invisible Things Lab to overcome this limitation. Blue Pill installs its thin hypervisor quietly without any intervention of the machine, especially no need to reboot the machine, and it moves the target OS into the VM by exploiting AMD-v extensions. However, by default it does not survive system reboot.

Although there are some security challenges, VM systems can provide compelling approaches to offering strong protection by the property of isolation. Garfinkel and Rosenblum [GR03] leveraged VMM technology and isolated an IDS from the monitored host by pulling the IDS outside of the host into a completely different hardware protection domain for greater attack resistance. They also offered VM introspection to inspect a VM from the outside for the purpose of analyzing the software running inside it. In addition, there are more and more hypervisors applied for secure system or security analysis. SecVisor [SLQP07] uses a small hypervisor to defend against kernel code injection, preventing an attacker from either modifying existing code in a kernel or from executing injected code with kernel privilege over the lifetime of the system. BitVisor [SET+09] is a parapass-through hypervisor that intercepts only a small set of I/O device access to implement OS-transparent data encryption and intrusion detection, while other access is mostly pass-through. HyperSafe [WJ10] endows existing hypervisors with a unique self-protection capability to provide lifetime control flow integrity. A lightweight hypervisor Hytux [ELND09] implements protection mechanisms in a more privileged mode than the Linux kernel and then protects the kernel from malicious actions.

### 10.4.3 Security Architecture Model

There has been some work on security architecture modeling and providing security architecture design guidance for architects [MQRG97, ZAF08, BDRS08]. However, none of them is specific for multicore systems, especially multicore architectures with hardware-assisted virtualization technology. The principle of separation kernel, introduced by [Rus81] in the early 1980s, is to divide all resources into blocks such that the activities in one block are isolated from activities in another block, unless an explicit communication has been established, and security policies are enforced by trusted applications running in some of those blocks. The conceptual model of separation kernel provides an ideal foundation to create secure VMMs, since the primary common functionality of both of them is to prevent illegal information flow between isolated blocks. Levin et al. [LIN06] extend and provide separation kernel protection profile to enforce a compound security policy with requirements at the gross partition level as well as at the granularity of individual subjects and resources, hence enhancing protection for secure systems.

Multiple Independent Levels of Security/Safety (MILS) [AFOTH06] is a high-assurance architecture for secure information sharing by separating out the security mechanisms and concerns into manageable components, such as separation kernels, partitioned file systems, and partitioning communications systems that deliver the required guarantee of separation.

# 11 EVALUATION OF HARDWARE FEATURES FOR SECURITY IN MULTICORE SYSTEMS

The evaluation of multicore hardware features is especially important when we begin to consider the use of these computer systems in high-security areas, with critical or secret data. A key feature of multicore architectures is virtualization technology. This chapter provides an overview of hardware extension support for virtualization to provide a context for later discussion of objectives. Intel and AMD have introduced hardware virtualization extensions in their current processors: virtual machine extensions (VMX) and secure virtual machine (SVM) extensions, respectively. Hardware-assisted virtualization technology provides several features to simplify VMM implementations for faster performance. When evaluating security about such a whole system, it is desirable to guarantee not just that the integrated system is secure, but also that separate concrete components are secure. Therefore, we will evaluate each component of hardware-assisted virtualization technology in current VM systems, including CPU virtualization, memory virtualization and I/O virtualization, based on the Intel Core i7 multicore architecture. In addition, because the CBEA processor is being considered for use in secure communication and data processing environment, we also need to explore its hardware architectural features prior to its utilization in these facilities. The details of each evaluation are presented in the following subsections.

## 11.1 CPU

For CPU virtualization, Intel VT-x introduces two modes of CPU operation: VMX root operation (a.k.a., host operating mode) that is provided for the VMM operation, and VMX non-root operation (a.k.a., guest operating mode) that is targeted for guest software execution. Both modes support execution in all four privilege levels, allowing OSs in VMs to manage processes as if they were on a stand-alone processor. By introducing both operating modes with full access to all privilege levels, the problem of ring compression disappears. This is because that a guest OS can operate in guest privilege level 0 while the VMM can still be fully protected from any errant behavior [FO06]. In addition, VT-x defines an in-memory data structure, referred to as *virtual machine control structure* or *VMCS*, to specify processor behavior as well as transitions between these two operating modes: the transition from VMX root operation to VMX non-root operation, *VM entry*; and the transition from VMX non-root operation to VMX root operation, *VM exit*. Processor behavior in VMX root operation is very much as it is outside VMX operation. The principle differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited. The guest OS instructions executed in VMX non-root operation are similar to the corresponding instructions in a non-virtualized system except for the fact that certain sensitive instructions and events cause VM exits to the VMM. The *VM-execution control fields* in the VMCS set the specific conditions for triggering a VM exit. To facilitate checking which events or exceptions caused a VM exit, the system records corresponding exit reason in a VM exit data structure once a VM exit occurs. Since these VM exits replace ordinary behaviors, the functionality of software in VMX non-root operation is limited. However, it is this limitation that allows the VMM to retain control of processor resources.

Figure 32 illustrates the VMX transitions that are involved during the execution of virtualized software. One or more guest VMs might be created on a single physical processor. Each guest VM is represented by a VMCS, which should be constructed before being launched. Before initializing a VMCS, system software needs to create a VMXON region in memory to support VMX operations for each logical processor. A VMM can host several VMs and have many VMCSs active under its management. A unique VMCS region is required for each VM while a unique VMXON region is required for the VMM itself. The address of the VMXON region is provided as an operand to the VMXON instruction. After executing the VMXON instruction, the VMM enters VMX root operating mode. Only in this operating mode can system launch guest VMs and then enter VMX non-root operating mode using the VMLAUNCH instruction. On VM entry, the host processor state is stored first and then the guest processor state is loaded from the constructed VMCS. In VMX non-root operating mode, the VMM regains control by affecting a VM exit. The VMM might finally decide to shut itself down and leave VMX root operation by executing the VMXOFF instruction.

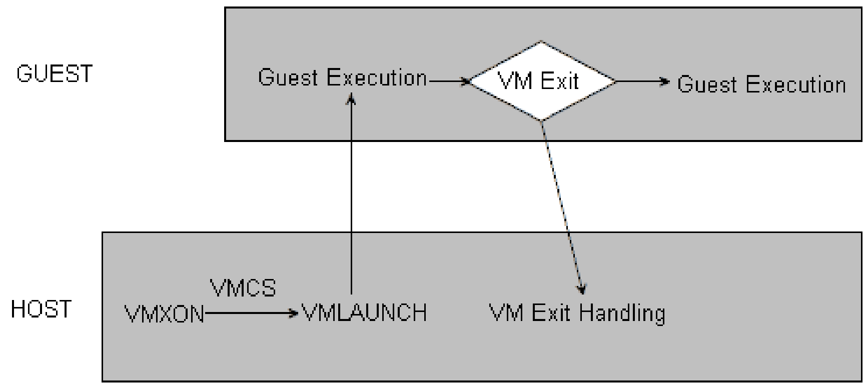


**Figure 32: Interaction between Virtual Machine Monitors and guests**

A VMCS is referred to as a controlling VMCS if it is the current VMCS on a logical processor in VMX non-root operation. A current VMCS for controlling a logical processor in VMX non-root operation may be referred to as a working VMCS if the logical processor is not in VMX non-root operation. The VMM can load a VMCS and make it the current VMCS by using VMPTRLD instruction. At any given time, only one VMCS is current for each logical processor. Figure 33 depicts a scenario swapping the execution between VM A and VM B in a logical processor. VMCS B is marked current after VMPTRLD B, and then the system can launch VM B while VMCS A is in an active state. After VM exit, the state of VMCS B is changed from controlling to current until VMPTRLD A is executed to make the VMCS B active. When VMCS A is current, VM A can be launched by using VMLAUNCH instruction.

The VMCS contains a guest state area, host state area and various VMX control fields that configure the environment of VM execution, VM exit and VM entry. However, in the AMD SVM extension, the VMCS-like data structure *virtual machine control block* or *VMCB*, only consists of guest state information, various control fields that configure the execution environment of the guest virtual machine and those fields that indicate special actions to be taken before running guest kernel code. This means that the host processor state information is not a component of the VMCB structure in AMD SVM and instead is memory-mapped to a region of memory with an initial starting address specified in the VM_HSAVE_PA MSR. Every field of the VMCS is associated with a 32-bit encoding, which can be provided as an operand to VMREAD or VMWRITE instructions when the hypervisor wishes to read or write that field. This type of access allows Intel to implement a VMCS according to the current layout of VMCS without causing compatibility problems with existing VMMs.

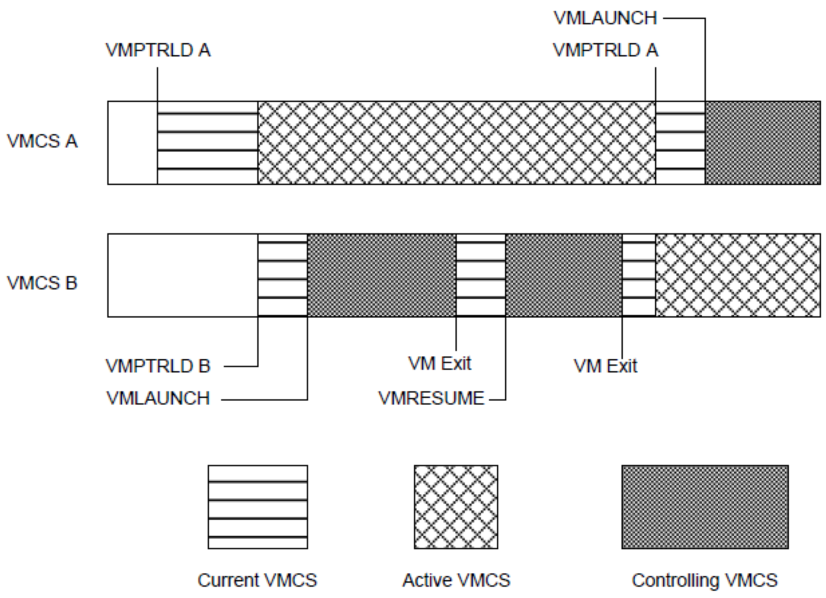


**Figure 33: State of VMCS and VMX Operation**

### 11.1.1 Sensitive but Unprivileged Instructions

According to Popek and Goldberg [PG74], classical virtualization requires that when a VM attempts to execute a sensitive instruction, it is guaranteed to trap automatically and signal the VMM. Back in the late 90's Robin and Irvine [RI00] examined the x86 architecture and found 17 sensitive but unprivileged instructions which don't meet this requirement. For example, the POPF instruction, one of sensitive and unprivileged instructions, pops a word from the top of the stack and stores the value in the lower 16 bits of the EFLAGS register, hence allowing values in the EFLAGS register to be changed that could impact the performance of another VM, thus violating classic virtualization concepts. Since Robin and Irvine's analysis, Intel has added over 500 new instructions through various language extensions. We examined these additional instructions and discovered 9 more instructions that also fall into this category: `MWAIT/MONITOR`, `XSAVE/XRSTOR`, `XSAVEOPT`, `RSM`, `XGETBV`, and `SYSCALL/SYSENTER`. Fortunately, Intel VT-x provides VM exit mechanisms to handle most of these instructions (We will evaluate these in Section13.2.1.1) if the VMM designer so chooses. Table 46 lists these 26 sensitive but unprivileged instructions and corresponding solutions in Intel VT-x. Appendix E provides details for all instructions that cause VM exits in Intel 64 architecture.

**Table 46: Corresponding settings for 26 sensitive but unprivileged instructions which cause VM Exits in Intel VT-x**

| *26 Sensitive but Unprivileged Instructions* | *Conditions to Cause VM Exits from VMX Non-Root Operating Mode in Intel VT-x* |
|---|---|
| SGDT, SIDT, SLDT, STR | "Descriptor-table exiting" VM-execution control is 1 |
| CALL, JMP, INT n | Task Switches not allowed |
| RET | IRET ("NMI exiting" and "virtual NMIs" VM-execution control are 1) |
| MOV | MOV instructions that load/store control registers, such as MOV to/from CR0/CR3/CR8; "user TPR shadow" VM-execution control is 1 |
| PUSHF(D)/POPF(D) | At the beginning of any instruction if RFLAGS.IF is 1 and "interrupt-window exiting" VM-execution control is 1 |
| PUSH/POP | Not caught in VM exit |
| LAR, LSL, VERR, VERW | Not caught in VM exit |
| MWAIT/MONITOR | "MONITOR exiting" or "MWAIT exiting" VM-execution control is 1 |
| RSM | If executed in SMM |
| XSAVE/XRSTOR/XSAVEOPT | Not supported in Intel Core i7-860. |
| XGETBV | Specifying a reserved or unimplemented XCR in ECX causes a general protection exception |
| SYSCALL/SYSENTER | SYSCALL is only supported in 64-bit mode |

The following provides a more detailed description of these extra 9 instructions why they belong to the category of sensitive but unprivileged instructions. `MWAIT/MONITOR` allows the hardware to set up a linear address range to be monitored. The processor stops executing instructions and enters an implementation-dependent optimized state while waiting for an event or a store operation to the address range armed by `MONITOR`, thus causing the processor to enter a different state. The instruction `RSM` returns control from SMM (System Management Mode[25]) to program or procedure, which affects the mode of the processor. `SYSCALL/SYSENTER` queries or changes the privilege level of the system by fast calling to ring 0 system procedures. `XGETBV` reads the XCR (Extended Control Register) which is a sensitive register. The instructions `XSAVE/XRSTOR` save/restore processor extended states to/from memory, thereby changing or referencing the processor state. `XSAVEOPT` which saves processor extended states optimized also changes the processor states. How these instructions are managed in our prototype IAVMM is illustrated in Section13.2.2.

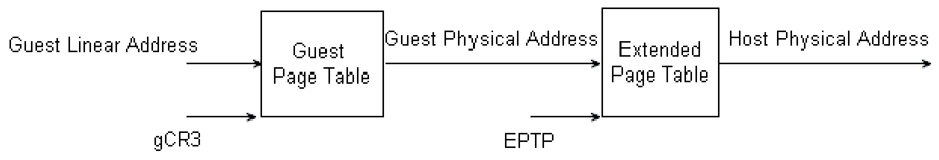
[25] SMM is a standard architectural feature in all IA-32 processors. This mode provides an OS or executive with a transparent mechanism for implementing power management features.

## 11.2 MEMORY

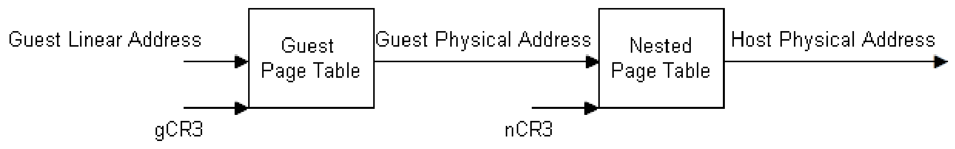
For the convenience of memory virtualization, Intel's Extended Page Table (EPT) and AMD's Nested Page Table (NPT) are proposed as second-level page tables in the VMM, accompanied by the guest page table (a.k.a. the first-level page table) in each guest OS, as described in Section 10.3.2. In addition, the following subsections also discuss shared memory access mechanism in the CBEA architecture.

### 11.2.1 Extended Page Table and Nested Page Table

As discussed in Section 10.3.2, the x86 architecture supports hardware translation of virtual addresses to physical addresses by use of the TLB. Entries in the TLB are updated by hardware that walks the page tables to find the relevant translation. The current root page table address is stored in control register 3 (CR3). Intel and AMD both virtualize CR3 to support VM memory management. Figure 34 presents address translations with Intel's EPT and AMD's NPT mechanisms. For Intel 64, guest linear/virtual addresses[26] are translated to guest physical addresses through guest paging structures. The guest paging structure is referenced by the guest CR3 which is used to locate the top level of the guest's hierarchical page table, such as page directory for each guest VM. Guest physical addresses are then translated to produce host physical addresses by traversing a set of EPT paging structures, the address of which is specified in the EPT base pointer field (EPTP) while constructing the corresponding VMCS. Likewise, AMD provides *gCR3* and *nCR3* for guest level and host level, respectively. The *gCR3* points to guest page tables (*gPT*) in guest physical memory which map guest linear addresses to guest physical addresses for each guest virtual machine, while *nCR3* points to nested page tables in system physical memory which map guest physical addresses to system physical addresses.



(a) Address Translation with Intel's Extended Page Table



(b) Address Translation with AMD's Nested Page Table

**Figure 34: Intel's EPT and AMD's NPT Mechanisms**

[26] Intel did not add segmentation support to its x86-64 implementation (Intel 64), so the guest linear addresses are treated the same as guest virtual addresses. However, code segments continue to exist in 64-bit mode and the segment base is treated as zero.

The original architecture for VMX operation required VMX transitions to flush the TLBs and paging-structure caches. This ensured that translations cached for the old guest linear address space would not be used after the transition. In order to solve this problem, a VM specific tag, virtual processor identifier (VPID), is introduced in the VMX operation by which a logical processor may cache information for multiple linear address spaces. This allows the TLB to keep track of which TLB entry belongs to which VM and the TLB entries of different VMs can coexist peacefully in the TLB, provided the TLB is big enough. As a result, guest software can be allowed to handle its own page faults, thereby reducing the frequency of VM exits and therefore avoid costly virtualization overhead. AMD-V also developed a similar VM specific tag, address space identifier (ASID), to aid in retaining cached information and making the logical processor switch to a different linear address space smoothly.

Using second-level page tables (EPTs or NPTs) associated with VM specific tag (VPID or ASID) has increased importance if there are multiple virtual CPUs per VM, because they have to synchronize the page tables many times with direct impact on the shadow page table update. With these tags in page tables, the CPU only has to synchronize TLBs as it would in a non-virtualized environment.

### 11.2.2 Shared Memory

As an example of another approach to memory virtualization, this section focuses on the IBM CBEA architecture. There are one PPE (PowerPC Processor Element) and 8 SPEs (Synergistic Processor Elements) in the CBEA processor (Figure 13(c)). Accesses to an SPE from all other system units (PPE, other SPEs, and I/O devices) are provided through memory mapped IO (MMIO) registers. With respect to logical partitions executing on the PPE, an SPE can be allocated to the logical partition by the hypervisor granting access to the SPE MMIO registers, which can be divided into three groups based on privilege [IBM07b]. Privilege 1 registers (most privileged) are used by the hypervisor to manage the SPE on behalf of a logical partition; Privilege 2 registers are used by the operating system in a logical partition to manage the SPE within the partition; and Privilege 3 registers (least privileged) are used by problem-state software, if direct access to the SPE from user space is supported by the operating system. Figure 35 presents the memory map of the CBEA processor's real-address space. The locations in the map are indicated by the sum of two values---an explicit offset added to a base real address. For example, "`0x080000`+*SPE1 BE_MMIO_Base*" means that the offset `0x080000` is added to the real address contained in the *SPE1 BE_MMIO_Base* base-address register.

In the CBE architecture, the DRAM memory system is shared among the threads concurrently executing on the PPE and each of the SPEs. According to Moscibroda and Mutlu [MM07], the memory system is "unfairly" shared among multiple cores, which results in degrading performance of another application running on the same chip. As a consequence, there exists a case that one SPE occupies a shared memory region and thus prevents other SPEs from accessing these resources efficiently. They call this aggressive application *Memory Performance Hog* (MPH), which could become a prevalent security issue that could affect almost all computer users. More details about experiments on this subject are provided in Section 13.1.2. Previous studies have shown that frequently multiple memory requests are clustered together and occur in a short time period. A number of memory scheduling schemes [RDK+00, NALS06, ZLZZ08] have been proposed to maximize memory bandwidth utilization and achieve the best overall performance for single-core processors, even for multicore processors. CBEA architectures apply row-hit-first memory access scheduling algorithm. This scheme schedules row buffer hits before misses to reduce the average memory access latency and improve bandwidth utilization.

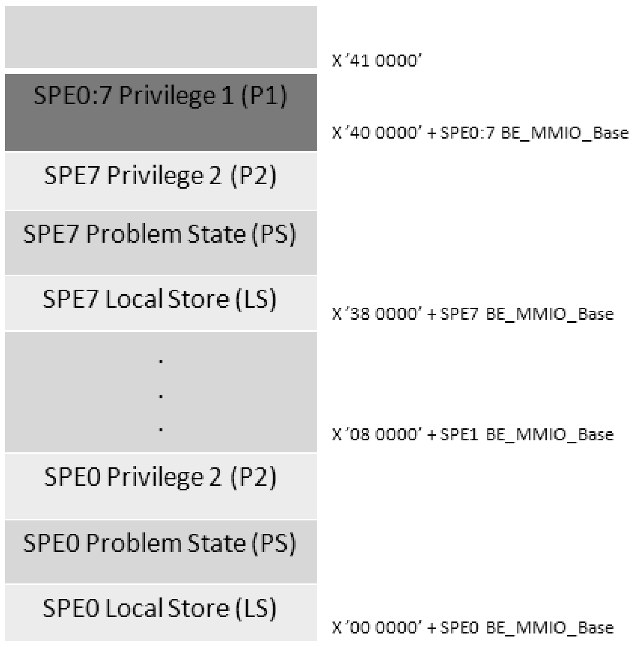


**Figure 35: CBE System Memory Map**

## 11.3 I/O DEVICES

Intel VT-d [Int08] and AMD-Vi (or IOMMU) [AMD09] extensions exist for I/O devices virtualization respectively since efficient I/O virtualization is an important consideration for hardware-assisted virtualization technology.

AMD's IOMMU enables hardware assisted memory management in the form of two facilities: the Graphics Aperture Remapping Table (GART) and the Device Exclusion Vector (DEV). The GART provides address translation of I/O device accesses to a small range of the system physical address space, and the DEV provides a limited degree of I/O device classification and memory protection that is essentially a table that permits or blocks DMA access between devices and memory pages. Intel's VT-d provides further functionality beyond that of AMD's DEV, such as interrupt virtualization. For example, devices on the traditional x86 could signal an interrupt using legacy I/O interrupt controllers or issue a message signaled interrupt (MSI) via DMA to a predefined address range, thereby violating the isolation property. Fortunately, VT-d redefines the interrupt-message format for MSIs by adding a message identifier and the hardware device's requester id in DMA writes to provide the necessary isolation [FO06]. Similar to IOMMU, VT-d helps the VMM better utilize hardware by improving reliability and security through device isolation using hardware assisted remapping, and improving I/O performance and availability using direct assignment of devices. VT-d incorporates software specified protection domains which restrict access only to dedicated devices assigned to a domain; unfortunately, it does not facilitate sharing a device among multiple guests, nor does IOMMU. A protection domain is abstractly defined as “an isolated environment to which a subset of the host physical memory is allocated" [AJM+06]. This implementation enables protection domains to be applied for virtual machines as well as device drivers running in the VMM.

## 11.4 EXTERNAL INTERRUPT

Intel VT-x allows both host and guest control of external interrupts through their own IDT. Host vectors refer to vectors delivered by the platform to the processor during the interrupt acknowledgement cycle. Guest vectors refer to vectors programmed by a guest to select an entry in its guest IDT. To meet the interrupt virtualization requirements, a VMM needs to take ownership of the physical interrupts and the various interrupt controllers in the platform. VMM control of physical interrupts may be enabled through the host-control settings of the "external-interrupt exiting" VM execution control. With guest control (external-interrupt exiting is clear), external interrupts do not cause VM exits and the interrupts delivery are masked by the guest programmed *RFLAGS.IF* value. With host control (external-interrupt exiting is set), external interrupts cause VM exits and are not masked by *RFLAGS.IF*. To take ownership of the platform interrupt controllers, a VMM needs to expose the virtual interrupt controller devices to the virtual machines and restrict guest access to the platform interrupt controllers.

How Intel VT-x handles interrupt processing in a VMM is illustrated in the following steps. First, the VMM sets up the guest to cause a VM exit to the VMM on external interrupts. This is done by setting the "external-interrupt exiting" VM execution control in the guest controlling-VMCS. Then, interrupts are automatically masked by hardware in the processor on VM exit by clearing RFLAGS.IF. If the VMM is utilizing the acknowledge-on-exit feature (by setting the acknowledge-interrupt-on-exit bit in guest VM-exit control field), the processor acknowledges the interrupt, retrieves the host vector, and saves the interrupt in the exit-interruption-information field (in the VM-exit information region of the VMCS) before transiting control to the VMM. Finally, the VMM can use the saved host vector to switch to the appropriate interrupt handler.

## 11.5 SECURE BOOTUP

AMD's SVM provides additional hardware support that is designed to facilitate the construction of trusted software systems. For example, the SKINIT (Secure Kernel Init) instruction reinitializes the processor to establish a secure execution environment for secure loader (SL) and then starts execution of the SL in a way that cannot be tampered with. The SL typically initializes SVM hardware mechanisms and starts a security VMM in a completely trustworthy manner, including setting up Device Exclusion Vector (DEV) protection for memory allocated for use by SL and VMM. The trusted platform module (TPM) receives the SL image sent by SKINIT and verifies the signature based on a secure hash comparison. The use of this special protocol provides a reliable means for verifying the startup of a trusted VMM. Similarly, the SENTER instruction, one of a few new instructions introduced by Intel Trusted Execution Technology (Intel TXT) [Int09b], also defines policy enforcement in hardware to block launch of unauthorized hypervisors, thereby allowing the hypervisor to protect itself against tampering. Intel TXT technology has provided a reliable method called "measured late launch" to load a clean hypervisor in a secure manner.

## 11.6 VIRTUALIZATION OF SYSTEM RESOURCES

When a VMM is hosting multiple guest environments (VMs), it must monitor potential interactions between software components using the same system resources. These interactions can require the virtualization of resources, which include debugging facilities, address translation, and physical memory. Before the description of this topic, we need to explain what resources are duplicated for each logical processor when Intel Hyper-Threading technology was introduced.

### 11.6.1 State of the Logical Processor

The following features are part of the architectural state of logical processors within Intel 64 processors supporting Intel Hyper-Threading technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)
- Segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS and EIP registers. Note that the CS and EIP/RIP registers for each logical processor point to the instruction stream for the thread being executed by the logical processor
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- Control registers and system table pointer registers (GDTR, LDTR, IDTR, task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, DR7) and the debug control MSRs
- Machine check global status (IA32_MCG_STATUS) and machine check capability (IA32_MCG_CAP) MSRs
- Thermal clock modulation and ACPI Power management control MSRs
- Time stamp counter MSRs
- Most of the other MSR registers, including the page attribute table (PAT). See the exceptions below.
- Local APIC registers.
- Additional general purpose registers (R8-R15), XMM registers (XMM8-XMM15), control register, IA32_EFER on Intel 64 processors

The following features are shared by logical processors:

- Memory type range registers (MTRRs)

Whether the following features are shared or duplicated is implementation-specific:

- IA32_MISC_ENABLE MSR (MSR address 1A0H)
- Machine check architecture (MCA) MSRs (except for the IA32_MCG_STATUS and IA32_MCG_CAP MSRs)
- Performance monitoring control and counter MSRs

### 11.6.2 Virtualization of System Facilities

The following system facilities are virtualized for virtual machines:

- Some MSRs. MSRs affect processor features, control the programming interfaces, or are used in conjunction with specific instructions. Intel VT-x virtualizes processor MSRs via VM execution controls (MSR bitmaps), VM-Exit controls, VM-Entry controls and some instructions (i.e. SYSENTER/SYSEXIT, SYSCALL/SYSRET, and SWAPGS).
- Debug facilities. In VMX operation, a VMM can support debugging a system and application software from within virtual machines if the VMM properly virtualizes debugging facilities. For example, the VMM can program the exception-bitmap to ensure it gets control on debug functions (e.g. breakpoint exception). The VMM may utilize the VM-entry event injection facilities to inject debug or breakpoint exceptions to the guest. MOV-DR exiting control bit for VM exit is implemented.
- Memory virtualization mechanisms. Memory virtualization allows the VMM to enforce control of physical memory and yet support guest OSs' expectation to manage memory address translation. Memory virtualization is required to support guest execution in various processor operating modes through the appropriate configuration of CR0 and CR4. VMX provides the hardware features required to fully virtualize guest virtual memory accesses, including allowing the VMM to trap guest accesses to the PAT (Page Attribute Table) MSR and the MTRR. Because a guest OS expects to perform normal memory management functions, it will access CR3, execute INVLPG, and modify page directories and page tables. Virtual TLB mechanism is provided for this purpose and avoids overhead.
- External interrupts. VMX operation allows both host and guest control of external interrupts. While guest control of external interrupts might be suitable for partitioned usages (different CPU cores/threads and I/O devices partitioned to independent VMs), most VMMs built upon VMX are expected to utilize host control of external interrupts. With host control of external interrupts, the VMM exposes software-emulated virtual interrupt controller devices (such as PIC and APIC) to each guest VM instance, such as directly mapping the physical device to the assigned VM.

The detailed virtualized registers are implemented by constructing VMCS for each VM. Appendix F gives a list of virtualized resources for each VM.

## 11.7 HARDWARE VIRTUALIZATION TECHNOLOGY SUMMARY

Based on the aforementioned analysis, Intel VMX and AMD SVM extensions both offer hardware assisted support for efficient virtualization in order to improve virtualization performance and reliability. Intel VT-x extensions are conceptually the same as those implemented in AMD-V (see Table 47) but many of the details differ, such as the fact that AMD64 memory controller is on-die with the CPU, while the Intel implementation is on a separate chip and may be implemented differently depending on the chipset used. There are still other small differences between them, e.g. interrupt virtualization.

**Table 47: Summary of HVT Support between Intel VT-x and AMD-V**

| | ***Intel VT-x (VMX)*** | ***AMD-V (SVM)*** |
|---|---|---|
| VM Data Structure | VMCS | VMCB |
| VM Extension | VM extensions (VMX) | Secure VM (SVM) |
| I/O Virtualization | Intel VT-d | AMD-Vi (a.k.a. IOMMU) |
| MMU Virtualization | Extended Page Table | Nested Page Table |
| Tagged TLB Entries | VPID | ASID |
| Secure Bootup | SENTER | SKINIT |

# 12 A LIGHTWEIGHT VIRTUAL MACHINE MONITOR FOR SECURE ANALYSIS OF MULTICORE ARCHITECTURES

With the trend towards multicore CPUs, it is naturally necessary for VMMs to support multiple cores. We are interested in a small VMM that will allow us to develop experiments that we could run in VMs and evaluate security features of the underlying Intel 64 hardware with hardware-assisted virtualization technology. Based on our review of existing VMMs (e.g. TVMM [Kan06] and MAVMM [NSJ+09]), we decided to create our own Intel-based Security Analysis VMM (IAVMM). In this chapter, we present our lightweight virtual machine monitor for Intel based machines, IAVMM, as well as the corresponding explanations for our architecture design. By taking advantage of hardware virtualization support and concentrating only on security analysis functionality issues, we were able to build a lightweight VMM and keep it thin and simple.

Our proof-of-concept VMM platform borrows some concepts from MAVMM project [NSJ+09] and extends it to support the Intel 64 architecture, as presented in an earlier paper [HAF11]. A more specific description of the prototype is discussed in detail in the following sections.

## 12.1 BOOTSTRAPPING

Our IAVMM executable is stored in a simplified 32-bit ELF format[27] readable by GRUB Legacy [GRU] that is used as a boot loader to start our system, can be booted up directly from USB flash drive (or external hardware) successfully, which means that it is a Type I VMM and actually boot on bare hardware machine. Once power is reset, after IAVMM is selected from the GRUB menu, GRUB starts in host mode and begins to load IAVMM kernel, which boots along with arguments such as memory map and command line parameters that are passed to IAVMM through a multiboot information structure.

Initially IAVMM enters root operating mode, then how to start a guest VM becomes a natural question. Three alternative ways were evaluated to pass the guest OS image to IAVMM for the guest VM. The first approach, similar to TVMM, is to use the *module* parameter specified in GRUB configuration and then execute this image directly after the *VMLAUNCH* instruction is called. Unfortunately, this requires the booting environment to be exactly as expected by the guest OS, which makes the configuration very complicated. The second is to execute GRUB in guest mode and then boot the guest OS. Address `0x7c00`, the beginning address of loaded master boot record which contains executable code of GRUB boot loader, is set in *GUEST_RIP* as guest initial instruction pointer address. Once the *VMLAUNCH* instruction is invoked, the guest instruction pointer points to `0x7c00` and switches to the GRUB boot selection menu. The third method is what we finally decide to use in IAVMM. We build a *guest_OS* image as the guest virtual machine, consisting of some research experiments for security analysis, and include it in our IAVMM image. In order to boot up the guest VM, VMM assigns the initial address of this guest OS image to *GUEST_RIP* during construction of the VMCS data structure, and

[27] We use the Xen tool *mkelf32* to build this ELF image from raw object files.

initializes the *HOST_RIP* field with the address of the function *VMX_VMExit_Handler*, to handle the VM exit procedure once a VM exit occurs.

### 12.1.1 AP Booting Procedure

One of our primary goals is to examine the potential security concerns among multiple cores. Therefore, at least one Application Processor (AP) as well as BootStrap Processor (BSP) are supposed to run in a VM system. In order to wake up the AP, the booting code must be relocated to low memory (first 1 MB of physical memory) since the AP can only be booted in real-address mode. Upon receiving an INIT IPI (Interprocessor Interrupt) from BSP, a local APIC causes an INIT at the AP processor. Then the BIOS causes the current processor to jump immediately to a specific location, where we store the code the processor will execute after waking up. This code switches the system processor to protected mode and jumps to the kernel. This small piece of code in low-memory is called trampoline code since it bounces the processor back up to higher memory. In order to perform this jump, system needs to process a warm-reset procedure. IAVMM first puts an appropriate pointer in the warm-reset vector[28], and sets the shutdown code by assigning address `0xf` (`0:f` in real-address mode) to `0xa`, and then causes an INIT IPI for the current processor to jump immediately to a specific location, where we store the trampoline code. The beginning address of the trampoline code must be page-aligned. IAVMM sets a bit to indicate which processor is currently running and clears any APIC error by writing a zero to the error status register. IAVMM can pass any parameters to the AP at this time if necessary.

By now IAVMM can actually start booting the AP processor, involving of sending a sequence of interrupts to the AP processor (see Figure 36). First, BSP sends an INIT IPI and asserts the INIT signal by writing the target processor's APIC ID to the high word of the ICR, then writes to the low word with the bits set to enable the INIT delivery mode, level triggered, and assert the interrupt. Deasserting INIT is not necessary for Intel Core i7. Then, IAVMM must wait 10ms to allow signals to propagate. Two STARTUP IPIs are needed to proceed in the next step with the routine. Clear APIC errors, set the target APIC ID in the ICR, and then send the interrupt by writing to the low word of the ICR with bits set for STARTUP delivery mode and with the code vector in the low byte. The code vector is the physical page number of the trampoline code, at which the processor should start executing. Wait 200μs, and then check the low doubleword of the ICR to make sure bit 12 is reset to indicate the message was dispatched before sending the second STARTUP. After these two STARTUP IPIs are sent successfully, the AP wakes up in real mode.

[28] IAVMM sets the warm reset vector at address 40:67 to the start of the trampoline code, which should reside in low memory.

```
BSP sends PA and INIT IPI
BSP delays 10ms
(BSP sends AP a deassert INIT IPI)
for (i=0; i<=2; i++){
  BSP sends AP a STARTUP IPI
  BSP delays 200μs
```

**Figure 36: Application Processor Startup Procedure**

## 12.2 OPERATING MODES

As usual, an Intel processor provides a set of memory-management tables that are designed to support basic system-level operations, including Global Descriptor Table (GDT) and Local Descriptor Table (LDT). These tables contain entries called segment descriptors, which provide the base address of segments as well as access rights, type, and usage information. When operating in protected mode, all memory accesses pass through either the GDT or the LDT. The GDT Register (GDTR) holds the base address and the 16-bit table limit for the GDT, so does LDTR. The base address specifies the linear address of byte 0 of the GDT; the table limit specifies the number of bytes in the table. Figure 37 depicts the GDT settings in IAVMM. Ring 0 entries are specified for the VMM level, whereas Ring 3 entries are particular for guest VM level[29]. Take the third entry (Index 2 shown in the table) as an instance. Because segmentation is disabled in long mode and code segments span all of virtual memory, the code segment base addresses are ignored. For the purpose of virtual address calculations, the base address is treated as if it has a value of zero, and the 20-bit segment limit ranges from 0 to `0xfffff` (1 MB). In this entry, the long (L) attribute bit and present (P) bit indicating that the segment referenced by the descriptor is loaded in memory are all set. The privilege level of this code segment (DPL, Descriptor Privilege Level) is zero.

```
    .globl gdt_table
    .align  PAGE_SIZE
 gdt_table:
        .quad   0x0000000000000000     /* null selector*/
        .quad   0x00cf9a000000ffff     /* ring 0 code, compatibility
                             (base =0, limit = ffff, R, P, G, D) */
        .quad   0x00af9a000000ffff     /* here is index 2 (host uses) : ring 0 code, 64-bit mode
                             (base =0, limit = ffff, R, P, G, L) */
        .quad   0x00cf92000000ffff     /* ring 0 data       R/W         */
        .quad   0x00cffa000000ffff     /* ring 3 code dpl = 3, compatibility E/R */
        .quad   0x00cff2000000ffff     /* ring 3 data        R/W        */
        .quad   0x00affa000000ffff     /* ring 3 code, 64-bit mode  E/R  (guest uses)*/
        .quad   0x00009a006000ffff      /* code 16 bit */
        .quad   0x000092006000ffff      /* data 16 bit */
        .quad   0x0000920b0000ffff      /* data 16 bit */
        .quad   0x008f89000000ffff     /* tss       */
        .fill   1, 8, 0           /* space for LDT per CPU     */
```

**Figure 37: Global Descriptor Table in IAVMM**

[29] Of course guest VM can be specified with Ring 0 Level. We set it to Ring 3 for research convenience.

Typically, VMMs transfer control to a VM using VMX transitions referred to as VM entries. The boundary conditions that define what a VM is allowed to execute in isolation are specified in a VMCS. In addition, processors may fix certain bits in CR0 and CR4 to specific values and not support other values, for example, the upper bits from bit 18 in CR4 are all reserved to 0. The first processor to support VMX operation requires that CR0.PE and CR0.PG be 1 in VMX operation. Thus, a VM entry is allowed only to guests with paging enabled that are in protected mode or in virtual-8086 mode. Guest execution in other processor operating modes needs to be specially handled by the VMM, such as guest execution in real-mode or in protected mode with paging disabled. As for the latter case, a VMM needs to use "identity" page tables to emulate unpaged protected mode in order to support such guest execution.

In IAVMM, both guest system and host system support 64-bit mode (long mode). Following a power-up or RESET initialization instructions, the processor is placed in 16-bit *real mode* first and processor resources are initialized to a known, consistent state, from which software can begin execution, such as the Bootstrap Processor's (BSP) fetching the first instruction that is located at physical-address `0xFFFF FFF0`. In this real mode, IAVMM loads the Interrupt Descriptor Table Register (IDTR) and Global Descriptor Table Register (GDTR) with pointers initially pointing to the corresponding data structures. Upon completion of establishing the real-mode environment, protected mode can be enabled by setting *CR0.PE* and then software can switch to *protected mode* environment. In protected mode, IAVMM initializes system data structures required by *long mode*. For example, IAVMM must be in protected mode with paging enabled before attempting to initialize IA-32e mode, and the long mode paging table must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the *MOV_CR3* instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode. As long as these long mode data structures are initialized and paging is disabled as well, software can enable and activate long mode by setting long-mode-active status bit *EFER.LMA* to 1. On the Intel 64 architecture, the *IA32_EFER* MSR is cleared on system reset. In general, switching the processor to long mode involves sequences like disabling paging (*CR0.PG* = 0), enabling physical-address extensions (*CR4.PAE* = 1), loading *CR3*, enabling long mode (*EFER.LME* = 1), and finally enabling paging (*CR0.PG* = 1). If these are not executed in this sequence, the system may crash. In long mode, the system-descriptor-table registers (GDTR, LDTR) continue to reference previous corresponding protected-mode descriptor tables. After switching to long mode, 64-bit operating systems should use the LGDT and LLDT instructions to load these registers with references to 64-bit descriptor tables.

## 12.3 MEMORY LAYOUT

Memory layout is a very important feature we need to manage due to the required isolation property of VM systems. Not all of the machine's physical memory address spaces are available to the system. Some memory is reserved by the BIOS, while some is used by ACPI[30], and so on. The information about which regions are available and which are reserved by the system is

[30] Advance Configuration and Power Interface (ACPI) specification provides an open standard for device configuration and power management by the operating system, which defines ACPI tables, ACPI BIOS and ACPI registers.

provided in a structure called e820 memory map[31] which can be queried for the future use. IAVMM only allocates necessary space in the available memory, which is in a BIOS-provided available physical RAM e820 range (see Table 48). As shown in the table, most of first 1 MB physical memory is reserved.

**Table 48: BIOS-provided available physical RAM E820 ranges (with 4 GB RAM)**

| Start | End |
|---|---|
| `0x0000 0000` | `0x0009 F800` |
| `0x000A 0000` | `0x000E 4000` |
| `0x0010 0000` | `0x7F77 0000` |
| `0x8000 0000` | `0xFEE0 0000` |
| `0xFEE0 1000` | `0xFFA0 0000` |

IAVMM allocates the low 128 MB memory for guest virtual machines while the host VMM occupies the last 32 MB of physical memory assigned to the host area (in the case of one VMM) as shown in Figure 38. There are two reasons for this memory layout and management. The first is that we can perform identity mapping between guest physical addresses and host physical addresses, thereby simplifying access to I/O. The second is that it is enough to load guest VMs in 128 MB space for our research experimentation.

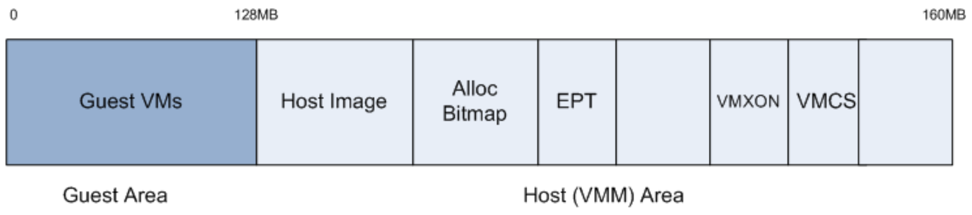


**Figure 38: Memory Layout of IAVMM**

### 12.3.1 Memory Layout in Single VM System

In the memory design of one VM system in our IAVMM, the first 128 MB memory is reserved for one guest VM and the image space of the IAVMM is allocated following this space. This means that the booting entry of IAVMM image would be the physical address `0x800000` or 128 MB decimal. The next section is set aside for Allocation Bitmaps area, consisting of host memory bitmap and guest memory bitmap. The bitmap adopts bit arrays to refer to a spatially mapped array of memory. Managing these two bitmaps separately allows us to protect host memory from the guest behavior. VMM heap area[32] starts right after Allocation Bitmap area. The first field dynamically allocated out of the heap is the EPT. The following region to be

[31] e820 refers to the facility by which the BIOS of x86 architecture reports the memory map to the operating sytsem or boot loader.

[32] The reason we identify this range as heap area is that we handle it dynamically if necessary. In other words, the Dynamic allocation mechanism of the heap area conveys that we shouldn't ask for memory space unless we need it.

allocated depends on whether or not a guest ELF image is configured. If an ELF image is specified as a module to the system, this area is allocated to load the ELF image. Otherwise it is allocated to other necessary structures for the VMM.

To address the needs of storing virtual machine state information, VMXON and VMCS areas are allocated for each logical processor to support VMX operations. The VMXON and VMCS pointers must be 4-KByte aligned. Before entering VMX operation, the host VMM must allocate a VMXON region, the physical address of which is provided in an operand to VMXON instruction. The VMM uses a separate VMXON region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor. A unique VMCS region is required to store data for each virtual machine and is used for communication and coordination between the VMM in root-mode and the VM in non-root mode. The VMM needs to prepare data fields in the VMCS that control the execution of a VM upon a VM entry. Before leaving VMX operation, a VMM is recommended to VMCLEAR all active VMCSs (those for which VMPTRLD has been executed), which ensures that all VMCS data cached by the processor are flushed to memory and that no other software can corrupt the current VMM's VMCS data. For management purposes, a VMCS revision identifier must be written to the VMXON region so as to enable the VMM to avoid using a VMCS region formatted for one processor on another processor that uses a different format. Once in VMX root operation mode, the system initializes some specific fields of the VMCS (using VMWRITE instruction) that guide execution of a VM upon a *VM entry*.

### 12.3.2 Memory Layout in Multiple VM Systems

In the above, we specified the memory layout in the case of a single VMM with one guest VM, shown in Figure 38. From the perspective of VMM, it can access the whole memory. However, the guest can access all the memory except the VMM region (128 MB - 160 MB), which protects the VMM region from modification by guest VM.

For the multicore system, we need multiple VMMs with multiple VMs, and the physical memory layout will be different because of the resource isolation property for multiple VM systems. To get an accurate view of the VM systems, Figure 39 presents the memory layout of two VMMs with two guest VMs. In this case, the first low 64 MB memory is shared by both of VM systems, wherein part of the first 1 MB is used to store the application processor booting procedure, since the application processor can only be booted from real-address mode. Since we are developing IAVMM for research experimentations, this shared memory would be accessible to all guest VMs and VMMs, and will not cause a security problem for our experiments. The following two 32 MB spaces are allocated to two guest VMs (GVM1 and GVM2), respectively. Two host VMMs occupy respective physical memory of size 32 MB starting from the physical address 128 MB, wherein the IAVMM image is located right after the guest space as single VM system does, which takes up 2 MB. Each of the four systems (GVM1, GVM2, VMM1 and VMM2) is allocated 32 MB memory.

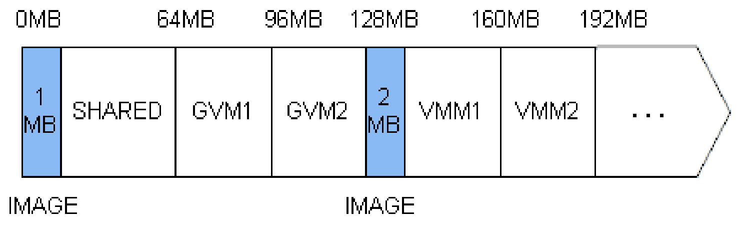


**Figure 39: Memory Layout in Multicore Systems**

From the perspective of memory protection, IAVMM needs to specify high privileged information invisible to guest VMs, hence protecting this sensitive information from being tampered with by the guest. From this point, we can reconfigure paging maps within the Extended Paging Tables (EPTs) to satisfy this security strategy so as to guarantee memory protection and offer better resource isolation. By building up the EPT appropriately, IAVMM redirects guest requests to access their own memory region (gray area), and hides the existence of other regions (blank area). Figure 40 and Figure 41 illustrate how this works for memory protection.

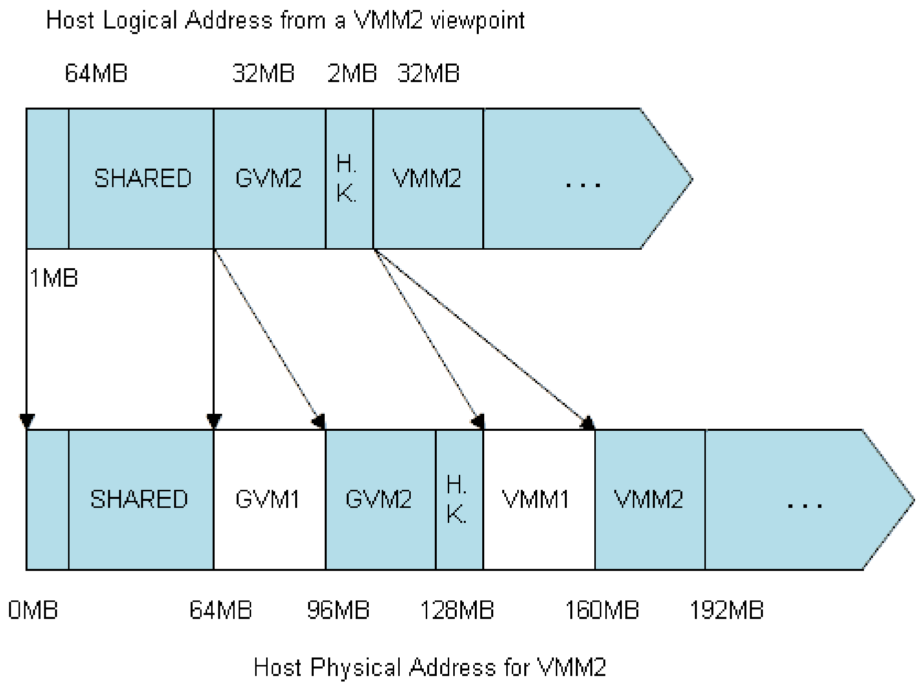


**Figure 40: Memory Translation from a VMM2 Viewpoint**

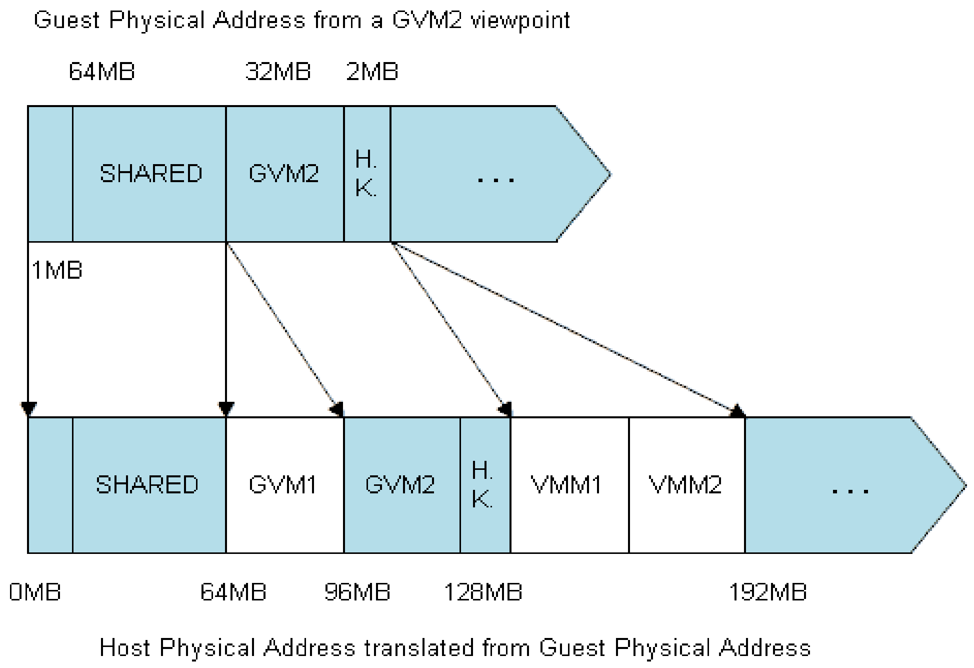

Guest Physical Address from a GVM2 viewpoint
64MB
32MB
2MB
SHARED
GVM2
H. K.
...
1MB
SHARED
GVM1
GVM2
H. K.
VMM1
VMM2
...
0MB
64MB
96MB
128MB
192MB
Host Physical Address translated from Guest Physical Address

**Figure 41: Memory Translation of EPT from a GVM2 viewpoint**

Figure 41 depicts the memory layout of EPT from the perspective of Guest VM2, which translates the guest physical address to the host physical address. The gray section in the figure is visible to the guest VM2 while the blank areas are not; that is, the guest VM2 is permitted to access the first low 64 MB shared memory, guest VM2 region, 2 MB host kernel (H. K.) area and the remaining spare area after the memory address of 192 MB. We can treat the host kernel the same as the image in the first low 1 MB, since both of them should be theoretically invisible to guest OS. IAVMM hides both guest VM1 and VMM areas from guest VM2. It seems that a guest VM is able to access a continuous block of memory, although the memory space is actually split into separate pieces of regions. From the VMM2 point of view, IAVMM also enables VMM2 regions available along with the aforementioned areas (gray sections in Figure 40). The same strategy is applied to VMM1 and corresponding guest VM1 as well. In this way, VMM2 can only access its corresponding guest VM (GVM2) rather than other guest VMs (e.g. GVM1), similarly with VMM1. Beyond the protection rings the hardware-assisted virtualization technology support, we provide this software solution to implement resource isolation to improve the reliability of TCB.

## 12.4 IAVMM PAGING SYSTEM

In our IAVMM prototype, we create host page table for the host VMM. The host page table covers the whole memory mapping including guest memory address and host memory address in IA-32e. Extended Page Tables are also built to support a second-level memory translation. In this section, we depict host page tables and EPT details, respectively, and illustrate the memory mapping with different page frames of size 2 MB and 4 KB.

### 12.4.1 Host Page Tables

As Figure 42 illustrates, host linear addresses are translated using a hierarchy of in-memory paging structures located using the contents of host CR3 which refers to the PML4 (Page Map Level 4) table. A 4-Kbyte naturally aligned *vmm_pml4_table* is set up as the PML4 table, which comprises 512 64-bit PML4E entries. The following 4 KB naturally aligned page-directory-pointer table (*vmm_pdp_table*) consists of 512 64-bit PDPTE entries. In the 2-MByte page frame case, *vmm_pdir_table* is initialized as the page-directory table with 512 64-bit PDE entries and the remaining 21-bits are allocated as the offset for a 2 MB page. In short, IAVMM memory mapping consists of 1 PML4E entry associated with 4 PDPTE entries, each of which maps 512 PDE entries with 2 MB page size per entry. Other unused entries in this 4-level paging structure are filled with zero. In this way, we have memory mapping of size 4 GB (4 GB = 1 * 4 * 512 * 2 MB). Similar with 2 MB page frame case, 4 KB page frame paging system just divides the last 21bits into two tables; the *vmm_pte_table* takes up the upper 9 bits, while the remaining 12-bit is allocated as the offset for a 4 KB page.

In our prototype, the guest page table is initialized to be identical to the host page table with the same format paging-structure entries, since we are just setting up the system for experiments.

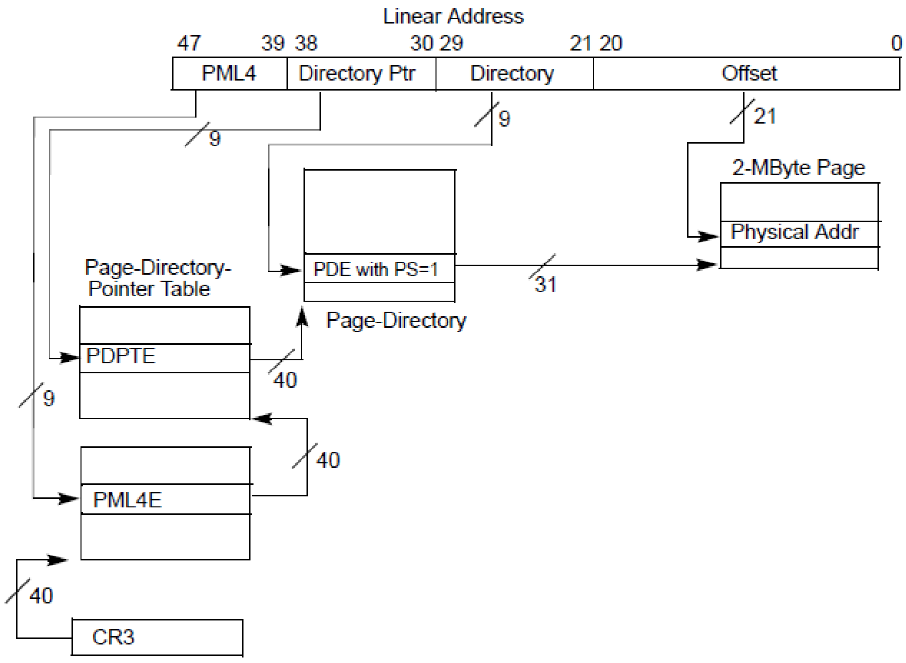


**Figure 42: Linear-Address Translation to a 2 MB page using IA-32e paging (Intel).**

### 12.4.2 Extended Page Table

The EPT mechanism is enabled when the flag "*enable EPT*" is set, which is controlled by secondary processor-based VM-execution control; that is, both the 31st bit flag "*activate secondary controls*" of the primary processor-based VM-execution control and the second bit flag "*enable EPT*" of the secondary processor-based VM-execution control need to be set to enable such mechanism.

The translation from guest virtual address to guest physical address is governed by the guest OS, while the EPT is handled by the VMM. If *CR0.PG* is set, the guest virtual addresses are translated using a hierarchy of in-memory guest paging structures located by the contents of *gCR3* (see Figure 34) and then guest physical addresses are translated by traversing a set of EPT paging structures to produce host physical addresses that are used to access memory. There are no guest paging structures if *CR0.PG* is clear; that is, each guest virtual address is treated as a guest physical address. With IA-32e paging, EPT pointer (EPTP) is specified to locate the first paging-structure, which is the physical base address of the Level 4 page map table (PML4) for the host physical address translation mechanism in our IAVMM system. IA-32e paging for Intel 64 processors may map virtual addresses to 4 KB pages, 2 MB pages, or 1 GB pages. IAVMM covers a structure case of either 2 MB page frame or 4 KB page frame. The EPT translation mechanism uses only bits 47:0 of each guest-physical address. It uses a page-walk length of 4, meaning that at most 4 EPT paging structure entries are accessed to translate a guest physical address. EPT in IAVMM maps all 4 GB memory except VMM region, which enable the system to hide VMM memory. A general-protection exception (GPe) is thrown when the guest tries to access this area.

The EPT in IAVMM system is created as shown in Figure 43 All of the 4 GB physical memory is mapped in 4-Level paging structure except the VMM region which is hidden to prevent it from tampering by guest OS. The prototype function *mmap_pml4(eptp, guest_paddr, host_paddr)* provides the paging mapping from guest physical address to host physical address. The implementation of *mmap_pml4()* function is in file *page.c*. It traverses deeply to build every EPT paging structure entry step by step from the top level PML4 through the bottom level PT if 4 KB page frame is to be built. The 4-Level is specified as *PGT_LEVEL_PML4, PGT_LEVEL_PDP, PGT_LEVEL_PD, PGT_LEVEL_PT*.

```
for (guest_paddr in all 4GB memory){
  Skip the necessary hidden VMM region;
  Identity memory map guest_paddr to host_paddr
    based on 4-Level Paging structure, e.g.
    mmap_pml4(eptp, guest_paddr, host_paddr);
```

**Figure 43: The procedure to create EPT**

As shown in Figure 42, these 48 bits are partitioned to traverse the EPT paging structures. A 4 KB naturally aligned EPT PML4 table is located at the physical address specified in bits 51:12 of the extended page table pointer (EPTP). An EPT PML4 table comprises 512 64-bit entries (EPT PML4Es). A 4 KB naturally aligned EPT page directory pointer table is located at the physical address specified in bits 51:12 of the EPT PML4E, which also comprises 512 64-bit entries (PDPTEs). Similarly, a 4 KB naturally aligned EPT page directory is located at the physical address specified in bits 51:12 of the EPT PDPTE, which comprises 512 64-bit entries (PDEs). In 4 KB page frame case, a 4 KB naturally aligned EPT page table (PTE) is located at the physical address specified in bits 51:12 of the EPT PDE.

Although the algorithm to build the paging structures of EPT and host page table is almost the same, there are still some conceptual differences between both of paging-structure entry's flags. While creating these page tables, these flags need to be configured as shown in Table 49. The first 3 bits in each entry flags have different meanings for host/guest page tables and EPT. In 2 MB page frame case, for host page table, *P*, *R/W* and *U/S* are set for *vmm_pml4_table* and *vmm_pdp_table* entries, while *P*, *R/W* and *PS* (Page Size bit) are set for *vmm_pdir_table* entries. However, for EPT, *R*, *W* and *E* are set for *pml4_table* and *pdp_table* entries, while *R*, *W*, *E* and *PS* (Page Size bit) are set for *pdir_table* entries.

**Table 49: Different Flags between host/guest page table entries and EPT entries**

| | Host/Guest Page Table Entries | EPT entries |
|---|---|---|
| Bit 0 | Present (P) | Read Access (R) |
| Bit 1 | Read/Write (R/W) | Write Access (W) |
| Bit 2 | User/Supervisor (U/S) | Execute Access (E) |

## 12.5 VM LAUNCHING PROCESS

There are two necessary steps to launch a guest VM. The first is to set up a VMM environment, and the second step is to launch the guest VM.

### 12.5.1 VMM Setup

VMMs need to ensure that the processor is running in protected mode with paging enabled before entering VMX root operation. The following list describes the minimal steps required to enter such an operation with a VMM running at CPL=0.

- Check if CPUID.01H:ECX.VMX[bit 5]=1 to determine that the processor has VMX support. *1H* is the input value for EAX, and ECX.VMX[bit 5] means that the output register *ECX* and field name with bit 5. The value *1* is the output.
- Create a VMXON region in non-pageable memory of a size 4-KBytes.
- Initialize the version identifier in the VMXON region with the VMCS revision identifier reported by *IA32_VMX_BASIC* MSR.
- Ensure the current processor operating mode meets the required CR0 and CR4 bits, such as CR0.PE=1, CR0.PG=1, and CR4.VMXE=1.
- Ensure that *IA32_FEATURE_CONTROL* MSR has been properly programmed and that is lock bit is set, which means that any WRMSR instruction to this MSR will cause a general exception.

Upon successful execution of VMXON instruction with the physical address of the VMXON region as the operand, the processor is in the VMX root operation. A VMM might leave VMX operation by executing VMXOFF instruction.

### 12.5.2 Handling of Guest VM Launching

The following list describes the minimal steps required by the VMM to set up and launch a guest VM.

1. Create a VMCS region in non-pageable memory of a size 4-KBytes.
2. Initialize the version identifier in the VMCS region with the VMCS revision identifier reported by *IA32_VMX_BASIC* MSR.
3. Execute the VMCLEAR instruction by supplying the guest-VMCS address and then the VMPTRLD instruction.
4. Construct a VMCS by issuing a sequence of VMWRITES to initialize various fields in the working VMCS, including various host-state fields, guest-state fields, and VM-exit control fields.
5. Execute VMLAUNCH to launch the guest VM.

Executing VMLAUNCH to launch the guest VM updates the controlling-VMCS pointer with the working-VMCS pointer and saves the old value of controlling-VMCS as the parent pointer (see Figure 33). In addition, the launch state of the guest VMCS is changed to "launched".

## 12.6 HANDLING OF VM EXITS

After a guest VM is running, the VMM can retain control through VM exit mechanisms. During construction of a VMCS for a guest VM, a number of VM execution control fields are provided to govern VMX non-root operation by specifying the instructions and events that cause VM exits. An exception results when a logical processor encounters an unusual condition that software may not have expected and then if the bit corresponding to the exception's vector is set in the exception bitmap, the exception causes a VM exit. When the condition causing an exception was established by the VMM itself, the VMM may choose to resume guest software after removing the condition. For example, the VM-execution control fields can set the conditions for triggering a VM exit, such as setting "*HLT exiting*[33]" and "*MWAIT exiting*[34]" in Processor-based VM-Execution Controls fields [Int09a]. After handling these VM exits, VMM software can resume guest software by VMRESUME instruction. With no change to VMCS, the VM will resume the same instruction that was executing when VM exit occurred. If the exit was to emulate a privileged or sensitive instruction, the VM exit handler must increment the program counter in the VMCS region before resuming.

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT in our IAVMM platform. For example, an exception causes a VM exit if the bit corresponding to that exception is set in the exception bitmap. Note the following:

- *EXCEPTION_BITMAP_MC*: If a machine-check exception happens.
- *EXCEPTION_BITMAP_UD*: If an UD instruction is executed.
- *EXCEPTION_BITMAP_PF*: If a page fault exception happens.
- *EXCEPTION_BITMAP_NM*: If a non-maskable interrupt happens.

[33] `HLT` exiting control determines whether executions of `HLT` cause a VM exit.
[34] `MWAIT` exiting control determines whether executions of `MWAIT` cause a VM exit.

The following lists the trigger conditions set in VM-execution control fields that would cause VM exits in our IAVMM platform.

- *NMI exiting*: When both of this control and the bit in the exception bitmap associated with exception's vector are set, non-maskable interrupts (NMIs) cause a VM exit;
- *External-interrupt exiting*: When this control is set and an external interrupt arrives, the MWAIT instruction causes a VM exit if this control is set;
- *MWAIT-exiting*: The MWAIT instruction causes a VM exit if this control is set;
- *MONITOR-exiting*: The MWAIT instruction causes a VM exit if this control is set;
- *MOV-DR exiting*: The MOV DR instruction causes a VM exit if this control is set;
- *WBINVD exiting*: The WBINVD instruction causes a VM exit if this control is set.

Once a VM Exit occurs, the exit reason will be recorded in the VM-exit information fields, which can be inspected by the VMM software. After the host decodes the reason for the exit, it can decide what to do next. In addition to the aforementioned controls, there are other instructions causing VM exits when they are executed in VMX non-root operation: `CPUID`, `GETSEC`, `INVD`, and `XSETBV`. This is also true of instructions introduced with VMX, which include the following: `INVEPT`, `INVVPID`, `VMCALL`, `VMCLEAR`, `VMLAUNCH`, `VMPTRLD`, `VMPTRST`, `VMREAD`, `VMRESUME`, `VMWRITE`, `VMXOFF`, and `VMXON`.

## 12.7 SUMMARY

We developed a lightweight VMM (IAVMM) that is designed especially for security analysis of Intel 64 architectural features. IAVMM does not implement unnecessary[35] virtualization features commonly found in general purpose hypervisors, such as Xen and VMWare. By taking advantage of hardware virtualization support and no need to use any Linux kernel APIs, we can make IAVMM thin and simple.

[35] We don't implement I/O and network.

# 13 EVALUATION EXPERIMENTS IN MULTICORE SYSTEMS

When investigating hardware security, it is critical to recognize that a multicore system is more than just a group of single cores. The researchers must consider current running programs accessing shared memory or I/O simultaneously as well as normal illegal memory accesses and commands. This interweaving of resources creates an uncertainty factor in protecting such a system. In order to get an accurate view of the multicore system, we first evaluate architectural features in the CBEA processor, and then examine hardware virtualization technology, one of the most important features in multicore architectures, especially in Intel Core i7 based on our IAVMM prototype.

## 13.1 EVALUATION OF CBEA PROCESSOR ARCHITECTURES

This section highlights some security concerns in the CBEA processor and possible solutions for some of them. Our group has provided a security review of the CBEA processor in a prior paper [SHAF10b]. In addition, we evaluated shared DRAM memory system mechanism in this architecture.

### 13.1.1 A Security Review of the CBEA

In this report, we focus on the Intel Nehalem architecture rather than the CBEA architecture; therefore we only list some major security concerns of the CBEA processor. Whatever the architecture brings, the memory system is always the most obvious target for attacks. For example, analyzing a potential covert channel in Multilevel Secure (MLS) systems is especially important, since covert channels are potentially uncontrolled data flows between process levels of security. In the architecture of a CBEA processor, each SPU contains a Local Store (LS) and 128 registers in place of traditional L1 and L2 cache system. The following provides a more detailed description.

The first issue posed is the security concern of the LS, a 256K cache-like memory assigned to each SPU. To allow programmers to manage concurrent data usage, the CBEA allows each SPU to access any other SPUs' LS through DMA read/write requests. While this greatly facilitates memory sharing, it makes keeping secure data on one SPU safe from other SPUs very difficult. For example, we can use one SPU (Eve) to read the communication between another SPU (Bob) and the PPU (Alice). Assume Messages, '*str*', are passed between Bob and Alice using a shared memory location, which resides at a static memory range. Then Eve can calculate its own *str* address and add or subtract the offset to find Bob's *str* address. Once Eve gets this address, she can read or overwrite the messages between Alice and Bob.

Second, the availability of the 128 registers to the programmer creates possible exploits as well. Of these registers, the first is a Link Register, the second a Stack Pointer, and the third an Environment Pointer. The first 80 Registers are for general use and are volatile, which means that they will be cleared as soon as the program terminates; registers numbered higher than 79 are the general use nonvolatile registers, which means that the data in these registers are not deleted if the context is not destroyed. Since they are memory mapped and accessible in the same manner as the SPU's LS, all of these registers create several possible security concerns. For instance, we can perpetrate a DoS (Denial-of-Service) attack on any SPU by merely writing a random memory address over the Stack Pointer and thus cause the SPU to abort. Following on the heels of this DoS attack is a flow redirection attack in which we no longer write a random memory address to the Stack Pointer but instead insert a location that contains another, unauthorized command. This is very similar to the traditional buffer overflow or format string attack. The CBEA does not provide separate memory protection to prevent this type of attack.

The open LS communication between SPUs prevents the development of the MLS system on the CBEA, and the DoS attack makes it impossible to guarantee the completion of a SPU program. Although the CBEA processor architecture provides some security features that can mitigate some of the aforementioned concerns, we do not believe that it is feasible to implement completely secure MLS systems. We identified several concerns that indicate the CBEA is not usable in a multi-level secure environment.

### 13.1.2 Evaluation of Shared DRAM System in CBEA

The arrival of multicore architectures creates significant challenges in the design of memory systems. In the CBEA, the DRAM memory system is shared among the threads concurrently executing on different processor cores. Due to the way current DRAM memory systems work, it is possible that a thread with a particular memory access pattern can occupy shared resources in the memory system, thereby preventing other threads from using those resources efficiently, which results in denial of memory service [MM07]. To explore this potential vulnerability, we evaluated the shared DRAM system to demonstrate whether such denial of memory service attacks exist in the DRAM system of the CBEA.

The modern DRAM system is a three-dimensional memory structure with the dimensions of bank, row and column. Each bank (see Figure 44) consists of multiple rows and columns. Consecutive addresses in memory are allocated for the consecutive columns in the same row. Each bank has one row-buffer and data can only be read from this buffer. The row-buffer contains at most a single row at any given time. Due to the existence of the row-buffer, once a particular location in a memory row is accessed, the entire row of the memory array will be transferred into the corresponding bank's row buffer. The row buffer, now combined with a sense amplifier, serves as a cache to reduce the latency of subsequent accesses to the same row. However, sequential accesses to different rows in the same bank result in high latency. Therefore, the memory access scheduler, the brain of the memory controller that selects a memory request from the memory request buffer to be sent to DRAM memory, is designed to maximize the bandwidth obtained from the DRAM memory. The CBEA employs a row-hit-first algorithm to select which request should be scheduled next. The row-hit-first policy requires that accesses to the same row in a bank are scheduled before the accesses to a different row even if those were generated earlier in time.

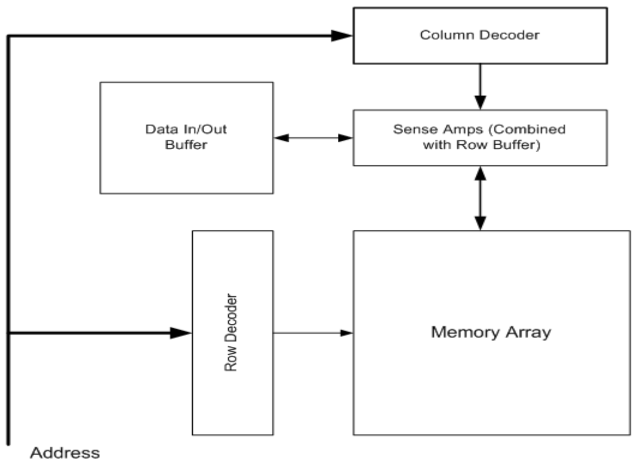


**Figure 44: A DRAM Bank**

In order to evaluate whether or not there exist potential DoS attacks in shared memory system according to Moscibroda and Mutlu's work [MM07], we built an experiment, in which all the SPEs issue DMA transfers to access the DRAM system concurrently, and record the execution time of each access. Figure 45 depicts the execution time of access to shared memory by each of the concurrently running cores. In the case of 2 cores, 4 cores and 6 cores, we found that DMA transfers take almost the same time for each core, which means that the row-hit-first scheduling algorithm does not degrade the performance of the system. Although the theoretical design of the CBEA has 8 SPE cores, there are actually only 6 active SPEs, with one SPE reserved for running the hypervisor and another one deactivated. Therefore, programs running on 8 cores actually have to swap in and out of the cores. This explains the fluctuation wave of the execution time appears in the 8-core case. From the above analysis, we conclude that the experiments in CBEA don't validate the "unfair" description of the shared memory system posed by Moscibroda and Mutlu (see Section 13.1.2), and we will not be concerned with this issue in CBEA any further.

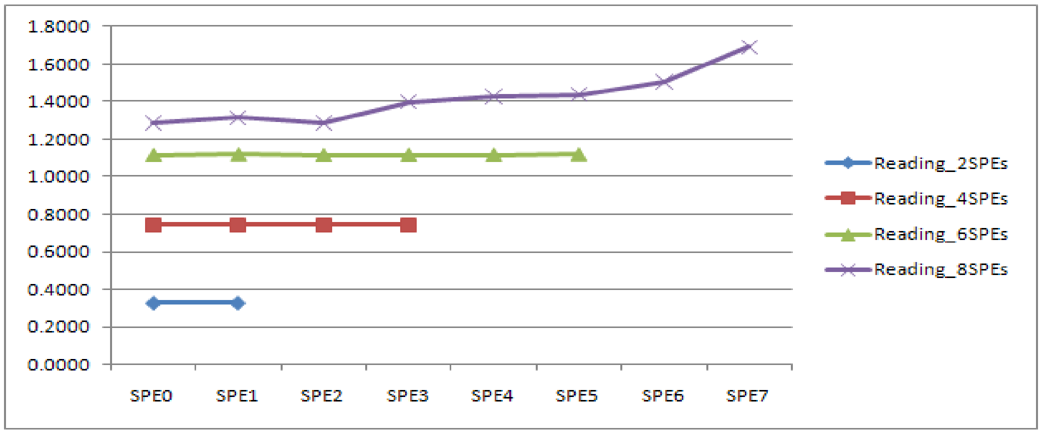


**Figure 45: Access execution time (μs) to shared memory when running multiple SPEs concurrently**

### 13.1.3 Discussion

The CBEA processor is designed for fast and free flow of information, not for security and restricted access. Although the unfairness of the shared DRAM memory system does not work in CBEA, it is of little surprise that some other security concerns arise over the memory system for such a processor, such as potential exploits in local store and registers. As mentioned previously, the current implementations of the CBEA processor have been examined in detail in our prior work [SHAF10b], and the results could serve to guide future security development of similar processors.

## 13.2 EVALUATION OF INTEL 64 HARDWARE FEATURES BASED ON IAVMM

The current system virtualization architecture allows a hypervisor to run an OS as a guest in a VM while maintaining control of the physical resources (e.g., CPU, RAM and devices). Such architecture inherently gives the hypervisor a complete view of system resources. The current popular tool for monitoring and analysis is virtual machine introspection (VMI), proposed by Garfinkel and Rosenblum [GR03], the process of examining a process inside a virtual machine from its hypervisor. While other works have leveraged this idea for security purposes, such as malware detection [JWX10] and intrusion detection [PSE09], our work focuses on hardware supported monitoring for security analysis based on IAVMM platform.

We evaluate Intel 64 architectural features based on our proof-of-concept IAVMM through simple security considerations, which include several aspects: systematic evaluation, instruction case studies and memory protection discussion. The extent of the system performance depends on how much the VMX extensions have been implemented in this platform. We built our experiments on a Intel-based hardware machine with a 2.80 GHz i7-860 processor that has Intel VT-x support and 4 GB of RAM. Without any support of system kernel APIs, IAVMM runs directly on the bare hardware, not requiring a host operating system, and the guest virtual machines lay directly above the hypervisor.

### 13.2.1 Systematic Evaluation

#### 13.2.1.1 IAVMM for Security Analysis of Bare Metal

From the perspective of the privilege rings model, VMMs can be classified into two types: Type I (running on the native hardware) and Type II (hosted). Robin and Irvine [RI00] concluded that it would be unwise to try to implement a high assurance VMM as a Type II VMM hosted on a generic commercial OS. Layering a highly secure VMM on top of an OS that does not meet reference monitor criteria would not provide a high level of security. A better approach would be to build a Type I VMM. Moreover, we need a hypervisor for security analysis of underlying Intel 64 architectural features. These are the reasons that we built the IAVMM based on the bare-metal architecture.

There are two advantages of IAVMM. First, IAVMM can provide a high degree of isolation between guest VMs. Second, existing popular OSs and their applications can run in this environment without modification. A VMM eliminates the needs to port software to a special secure platform and supports the functionality of current application suites. However, the biggest disadvantage to a Type I VMM approach is that device drivers must be written for every device to ensure the virtualization. Although devices are not currently supported in IAVMM and it avoids this problem, we will consider it in the future, if necessary.

#### 13.2.1.2 Lightweight

The system which we want is a simple and convenient hypervisor for security analysis. Initially, one of the reasons for this was purely practical. Implementing more code in the hypervisor leads to more bugs being added to the hypervisor, which often crashes the development machine and is also much more difficult to debug. Therefore, VMMs are considered more trustworthy than traditional OSs because of their smaller TCB (Trusting Computing Base). Compared with commodity hypervisors such as Xen and VMware, IAVMM is really a lightweight hypervisor as we expected. It is a simple and convenient system for security analysis of Intel 64 architecture. It does not provide full functionality of the commodity VMMs, but enough for our experimentation, and is extensible if we need more features.

Table 50 summarizes the features of different VMMs. IAVMM only consists of 7K lines of code and thus is convenient for us or other researchers to understand and develop further, such as applying necessary security policies as we will. Our code base is 3 orders of magnitude smaller than commodity VMMs, such as Xen and VMWare ESX. TVMM and MAVMM are lightweight and Type I (bare metal) VMMs as well, but both of them only support the AMD architecture and need to boot in the AMD SimNow Simulator. MAVMM only supports booting 32-bit OSs and guest systems which may expect to start in real or legacy protected mode with paging off, whereas IAVMM is able to boot the host and the guest in 64-bit mode (long mode). TVMM does so but merely with a guest paging structure of 2 MB pages, and our system provides a page structure of either 2 MB or 4 KB pages.

**Table 50: Systematic Features of Various VMMs**

| | ***Xen 3.4.1*** | ***VMWare ESX*** | ***TVMM*** | ***MAVMM*** | ***IAVMM*** |
|---|---|---|---|---|---|
| Code Size (Lines) | 230K | 200K | 4K | 9K | 7K |
| Hardware Platform | Intel/AMD | Intel/AMD | AMD | AMD | Intel |
| Operating Mode | 32/64 bits | 32/64 bits | 64 bits | 32 bits | 64 bits |
| VMM Type | Type I | Type II | Type I | Type I | Type I |

The size of TCB (Trusted Computing Base) is an important factor to consider when evaluating a system's security. Our proof-of-concept IAVMM has a lightweight VMM kernel and makes it easier to avoid bugs, and to formally verify desired properties of the system.

### 13.2.2 Instruction Case Studies

#### 13.2.2.1 Sensitive but Unprivileged Instructions

Hardware virtualization requirements indicate that there must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction[36]. Back in the late 1990s, Robin and Irvine [RI00] found 17 sensitive but unprivileged instructions falling into this category.

[36] Sensitive instructions are those that change the configuration of the system, or whose behavior depends on the configuration.

Since that time, Intel has added over 500 new instructions through various language extensions. We examined these additional instructions and discovered 9 more instructions that don't meet that requirement (classical virtualization) as well, including MWAIT/MONITOR, XSAVE/XRSTOR, XSAVEOPT, RSM, XGETBV, and SYSCALL/SYSENTER. These 26 total sensitive but unprivileged instructions are listed in Appendix D. Fortunately, Intel VT-x provides VM exit mechanisms to handle most of these instructions, if the VMM designer so chooses. The experiments of all these instructions are built as follows:

- **`MWAIT` and `MONITOR` Instructions:** Whether instructions `MONITOR` and `MWAIT` are supported in this architecture or not depends on whether the `MONITOR, CPUID` feature flag (CPUID.01H:ECX.MONITOR[bit 3]) is set. Both instructions are used together to monitor a range of linear memory. The MONITOR instruction sets up an address range for the monitor hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using `CPUID`) and puts the monitor hardware in armed state. The execution of `MWAIT` is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by the preceding `MONITOR` instruction in program flow, halting most activity in the core while doing so.

  An interrupt will cause the processor to exit if ECX = 1 within `MONITOR` instruction, and then the execution will resume at the instruction following the `MWAIT` instruction. If the ECX within `MWAIT` is equal to 1, then triple fault will be thrown out. Most of VMMs don't have such an exception handler to process this triple fault and ignore these executions or stall the system, which means that the VMM doesn't execute these instructions as expected by the guest VM. Fortunately, Intel VT-x guarantees that the `MWAIT` instruction would cause a VM exit if the "MWAIT exiting" VM-execution control is set (see Figure 46). Once instructions `MONITOR` and `MWAIT` are invoked in a guest VM, the event causes a VM exit and then the VMM will determine how to handle these two sensitive but unprivileged instructions. We tested this and found that we could capture the instructions and corresponding triple fault without a problem.

```
...
guest CR0: 0x80040033, CR3: 0x800a000, and CR4: 0x2028
launching guest for CPU 0x0

-------------Guest VM in BSP: hello guys-------------
Vmx_vmexit_handler, exit_reason=0x24
MWAIT or MONITOR is called in Guest VM
VM exit handling done,
then stall here
...
```

**Figure 46: IAVMM traces CR information while launching a mini_guest**

- **SYSENTER and SYSEXIT Instructions**: Companion instructions SYSCALL/SYSRET and SYSENTER/SYSEXIT were independently created by AMD and Intel, respectively, but in essence do the same thing. SYSENTER executes a fast call to a level 0 system procedure or routine, and is a companion instruction to SYSEXIT, although they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code. Far calls to different privilege levels are problematic because they involve the CPL, DPL and RPL. If a far call is executed to a different privilege level, accessing the code segment for the procedure has to be through a call gate. A task uses a different stack for every privilege level. Therefore, when a far call is made to another privilege level, the processor switches to a stack corresponding to the new privilege level of the called procedure. Neither the SYSENTER and SYSEXIT instruction supports passing parameters on the stack. Executing these instructions has many privilege level checks that compare the CPL and RPL to DPLs. Since the guest software normally operates at user level (CPL 3), these checks will not work correctly when it tries to access guest OS at CPL 0.

  Given our experiments based on IAVMM, when the system is in VMX non-root operating mode, guest software is running at privilege level 3, but without a guest OS running concurrently. Therefore, using these instructions for transitions between guest software code and guest OS procedures in IAVMM will cause triple fault. In general, when both guest software and guest OS exist in a guest VM system, there may be a security hole. Certainly, SYSENTER can be called from anywhere in guest software code, and SYSEXIT is expected to return to where called. Since the user can forge a return address, which might be an address to a malware virus, the execution of SYSEXIT can cause unexpected results. Fortunately, in VT-x, modifications of this return address require to read and write *IA32_SYSENTER_EIP* and *IA32_SYSENTER_ESP* MSRs, but operations of MSRs ( i.e. RDMSR and WRMSR) in guest VM will cause a VM exit.

- **XGETBV Instruction:** The XGETBV instruction reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. This instruction provides convenience for reading system registers and should be documented as a crucial security feature, however, on the Core i7-860 processor, the high-order 32 bits of RCX, RAX, and RDX are ignored. Currently, only XCR0 is supported, which is the *XFEATURE_ENABLED_MASK* register. Thus, all other values of ECX are reserved and will cause a general exception.

  Once executing this instruction, IAVMM will produce a general exception trapped with invalid opcode (6) as shown in Table 51.

- **XSAVE, XRSTOR, XSAVEOPT, and RSM Instructions:** The XSAVE instruction performs a full or partial save of enabled processor state components to a XSAVE save area, the memory address of which is specified in the destination operand. Both XSAVE/XRSTOR and XSAVEOPT are not supported on Intel i7-860 processor. The RSM returns program control from SMM to the application program or operating system procedure that was interrupted when the processor received an SMM interrupt. Since there are no XSAVE, `XRSTOR`, `XSAVEOPT` instructions on Intel Nehalem architecture and the guest VM system can't be switched to SMM mode, the experiments of all these instructions cannot be built in IAVMM.

**Table 51: Selective Exception Bitmap**

| *Vector No.* | *Description* | *Error Code* | *Source* |
|---|---|---|---|
| 0 | Divide Error | No | DIV and IDIV instructions. |
| 2 | NMI Interrupt | No | Nonmaskable external interrupt. |
| 3 | Breakpoint | No | INT 3 instruction. |
| 4 | Overflow | No | INTO instruction. |
| 5 | BOUND Range Exceeded | No | BOUND instruction. |
| 6 | Invalid Opcode (Undefined opcode) | No | UD2 instruction or reserved opcode. |
| 8 | Double Fault | Yes | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | Coprocessor Segment Overrun | No | Floating-point instruction. |
| 10 | Invalid TSS | Yes | Stack switch or TSS access. |
| 12 | Stack Segment Fault | Yes | Stack operation and SS register loads. |
| 13 | General Protection | Yes | Any memory reference and other protection checks. |
| 14 | Page Fault | Yes | Any memory reference. |
| 17 | Alignment Check | Yes | Any data reference in memory. |
| 18 | Machine Check | No | Error codes (if any) and source are model dependent. |

- **`SGDT`, `SIDT`, and `SLDT` Instructions:** All three of these instructions, `SGDT`, `SIDT`, and `SLDT`, store a sensitive register value into some specific location, but they are not privileged in the Intel architecture. Since the Intel processor only has one `LDTR`, `IDTR`, and `GDTR`, a problem arises when multiple guest VMs try to use the same registers for their own operations. This means that VM2 can retrieve the contents of these registers in VM1. VT-x handles this problem by adding "descriptor-table exiting" bit in VM-execution control to cause VM exit once one of these three instructions is invoked. The experiments demonstrate that when guest software attempted to execute `SGDT`, `SIDT`, and `SLDT` and the "descriptor-table exiting" VM-execution control was 1, a VM exit occurs.

- **`STR` Instruction:** The `STR` instruction stores the segment selector from the task register into a general-purpose register or some memory location. If a VM runs at a run-time privilege level of 3 it can use this instruction to retrieve the contents of the STR, which may have been modified by the OS running at a higher level. Therefore, VT-x handles this problem in the same manner as the `SGDT` instruction, when "descriptor-table exiting" VM-execution control was 1, a VM exit occurs.

- **LAR, LSL, VERR and VERW Instructions:** The LAR instruction loads access rights byte from a segment descriptor into a general-purpose register and sets the ZF flag in the *EFLAGS* system register if the load operation is successful. The LSL instruction loads the unscrambled segment limit from the segment descriptor into a general-purpose register and sets the ZF flag in the *EFLAGS* system register if the load operation is successful. The VERR and VERW instructions verify whether a code or data segment is readable or writable from the current privilege level, respectively. The execution of any of these four instructions within a guest VM results in the modification of sensitive resource to lower privilege level when the VM runs at privilege level 3. The experiments demonstrate that the execution of all these four instructions do not cause VM exit and any exceptions.

- **PUSH and POP Instructions:** The POP instruction loads the value from the top of the stack to the location specified with the destination operand, which may be a general-purpose register, memory location, or segment register. The PUSH instruction has the opposite effect of the POP instruction. The experiments demonstrate that the execution of both instructions do not cause VM exits and any exceptions.

- **PUSHF and POPF Instructions:** The PUSHF and POPF instructions reverse each other's operation. The PUSHFD and POPFD instructions are the 32-bit counterparts of the PUSHF and POPF instructions. Both of them are sensitive but unprivileged instructions, since the PUSHF instruction pushes the lower 16 bits of the *RFLAGS* register onto the stack, which allows modification of certain bits in the *RFLAGS* register that control the operating mode and state of the processor. Fortunately, Intel VT-x addresses this problem. A VM exit occurs at the beginning of executing either instruction if *RFLAGS.IF* is 1.

- **MOV to/from CR Instructions:** The experiment demonstrates that moving the contents of a control register (*CR0*, *CR2*, *CR3*, *CR4*, or *CR8*) to a general-purpose register or the contents of a general purpose register to a control register can cause a VM exit, since loading and storing control registers are sensitive to the system state. The basic exit reason shows guest software attempted to access control registers.

- **CALL, JMP, and INT n Instructions:** All three of these instructions attempt a task switch, which is not allowed in guest VM, according to VT-x. For example, the INT n instruction performs a call to the interrupt or exception handler specified by n. It pushes the RFLAGS register onto the stack before pushing the return address. The execution of this instruction in VT-x causes a VM exit if the "NMI exiting" is set.

- **RET Instruction:** The RET instruction transfers program control to a return address that is placed on the stack, whereas the IRET instruction returns control from an exception, interrupt handler, or nested task. If "NMI exiting" is set, once such an instruction is executed, a VM exit occurs.

From the aforementioned experiments, Intel VT-x provides mechanisms to handle most of these 26 sensitive but unprivileged instructions smoothly, the execution of which cause VM exits if corresponding settings are satisfied. However, there still are 6 instructions `LAR`, `LSL`, `VERR`, `VERW`, `PUSH` and `POP`, which do not generate VM exits and remain sensitive.

#### 13.2.2.2 Undefined Instruction

The `UD2` instruction generates an invalid opcode, which is provided for software testing to explicitly generate an invalid opcode. Other than raising such an invalid opcode exception, this instruction is the same as the `NOP` instruction.

In IAVMM, we created a VM exit handler, which handles exceptions occurring from a guest VM. Therefore, once a `UD2` instruction is executed, an exception (GPe) will be reported and the handler will deal with the exit. However, there is no exception handler implemented in VMM level[37]. Thus, all of the processors will be halted, including the other running processors concurrently. If we create exception handler for this on the VMM level, this exception can be caught and then system can transfer the control back to the caller program flow.

#### 13.2.2.3 Bitmap

To support VMM flexibility, the VMCS adds bitmaps that allow VMM conditions' selectivity regarding to most VM exits. The following items detail three of these:

- *Exception Bitmap*: This bitmap is a 32-bit field that indicates one bit for one entry for the x86 exceptions. It allows a VMM to specify which exceptions should cause VM exits and which should not.
- *I/O Bitmap*: This bitmap contains one entry for each port in the 16-bit I/O space. An I/O instruction causes a VM exit if it attempts to access a port whose entry is set in the I/O bitmaps.
- *MSR Bitmap*: This bitmap contains two entries (one for read, one for write) for each MSR currently in use. The execution of `RDMSR/WRMSR` will cause a VM exit if it attempts to read/write an MSR, whose read/write bit is set in the MSR bitmap.

Take the exception bitmap for an instance, if an exception occurs, its vector indicates whether the corresponding bit in the exception bitmap is set. If this bit is set, a VM exit is triggered; otherwise, the exception is delivered through the guest IDT. See Table 51 for details. The Error Code column indicates whether an error code is saved when the exception occurs.

Page faults (exceptions with vector 14) are a little bit different. When a page fault occurs, whether it causes a VM exit or not is determined not only by bit 14 in exception bitmap, but also by the error code *PFEC* produced with the page fault and two 32-bit fields in the VMCS (the page-fault error-code mask field *PFEC_MASK* and page-fault error-code match field *PFEC_MATCH*). If *PFEC & PFEC_MASK = PFEC_MATCH*, the specification of bit 14 in the exception bitmap is followed; that is, a VM exit occurs if that bit is set. If not, the meaning of

[37] Because this platform is for research experimentations, and doesn't provide full functionality of commodity hypervisors. However, exception handler for guest VM is implemented.

that bit is reserved; that is, a VM exit occurs even if that bit is clear. Thus, if the design requires VM exits on all page faults, software need to set the bit 14 in the exception bitmap and set the *PFEC_MASK* and the *PFEC_MATCH* each to `00000000H`, which is exactly what IAVMM does. If page faults are not required, software just needs to set *PFEC_MATCH* to `FFFFFFFFH`.

### 13.2.3 Covert Channels Analysis

A covert channel is a method of passing messages in an unanticipated or unintended manner [U.S93], thus could allow a potentially uncontrolled data flow among different cores in a multicore system. There are certainly various unexplored covert channels hiding in multicore architectures, and it is important to be aware of their existence. According to US Department of Defense Policy [U.S93], in secure systems, indirect, covert information flows should be identified, measured and monitored at run-time. Moving on from this issue, we have begun to explore different potential covert channels given Intel's current implementations. Our goal is to evaluate and document these and provide guidance for developers on possible software work around. The following lists two examples of these types of issues: Processor Caches and Registers.

- **Processor Caches:** For processors supporting Intel Hyper-Threading Technology, the processor caches are shared among logical processors in one physical core. Any cache manipulation instruction that is executed on one logical processor has a global effect on the cache hierarchy of the physical processor.

  Each logical processor has its own control register CR0, and thus its own CD (Cache Disabled) flag in *CR0*. The CD flags for the two logical processors are ORed together, such that when any logical processor sets its CD flag, the entire cache is nominally disabled, thereby degrading the performance. This degradation can easily form the basis for a covert channel.

  Several instructions, including `WBINVD`, `INVD`, and `CLFLUSH` instructions, might invalidate the entire cache hierarchy of two logical processors. For example, by using WBINVD instruction, the system invalidates the entire cache hierarchy after modified data is written back to memory. All logical processors are stopped from executing until after the write-back and invalidate operation is completed. As a consequence, the use of the WBINVD instruction can have an impact on interrupt/event response time.

- **Registers:** In this section we explore several registers that might impact resource isolation, such as loading a task register and `MOV_DR`.

  The processor state information, which is used for restoring a task, is stored in a system segment called task-state segment (*TSS*). The *TSS* is defined by a segment descriptor that may only be placed in the *GDT*. The *busy* (B) flag in the field *type* of a *TSS* descriptor indicates whether the task is currently running or suspended; the *type* field with a value of 0x9 (1001B) indicates an inactive task and a value of 0xb (1011B) indicates a busy task. To insure that there is only one *busy* flag associated with a task, each *TSS* should have only one *TSS* descriptor that points to it. In most systems, the DPL (Descriptor Privilege Level) fields of *TSS* descriptors are set with privilege level value less than 3, so that only privileged software can perform task switching. In IAVMM, we build one VMCS for each guest VM in a single core, where we need to load task register. After loading a non-zero host *TR* for one VMCS to mark it busy, we must clear the *busy* flag to avoid blocking other APs to construct their own VMCSs. Otherwise other APs can’t load their task register successfully

`MOV_DR` indicates that the caller accesses a debug register which is used by a processor for program debugging. There are eight debug registers totally, DR0-DR7, with DR4 and DR5 obsolete synonyms for DR6 and DR7. These debug registers allow programmers to selectively enable various debug conditions and hold the addresses of memory and I/O locations called breakpoints. A set of four debug addresses are specified in DR0-DR3 as breakpoint locations, and conditions can be defined in debug control register DR7. They are set to the memory address where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. However, one vulnerability behind this instruction is that it does not check guest CPL (Current Privileged Level) before VM Exit; that is, we can set the debug registers without a privilege level 0 when we are in VM mode. This can easily be used for DoS attack because we can just make the guest kernel double fault. Therefore, when some breakpoints in DR0-DR3 are set, once a debug exception is generated, the execution of instructions would be stalled no matter what privilege level programs are running at. We have not found a software workaround for this problem yet.

### 13.2.4 Memory Access Discussion

#### 13.2.4.1 Guest OS trying to access host region

In this test suite, we demonstrate the memory protection mechanism on IAVMM, which prevents guest software from accessing the host memory region. When we build the EPT for IAVMM, the VMM region is hidden by skipping the VMM region within the EPT memory mapping from the perspective of the guest. Therefore, under this protection mechanism, guest software is prohibited from affecting or modifying the execution of the VMM.

One test case is designed for guest software to access a VMM region, comparing with the other test case for guest software to access its corresponding guest region. The experiment is implemented as follows. First, move guest OS procedure into low memory (2 MB) which must be within the guest memory region. This is because our executable code is initially stored in the high memory (128 MB) which is not a designated guest memory area, we need to move that code to the guest space where it is supposed to be. Second, issue a JMP or CALL instruction from guest executable code to access a function belonging to the VMM region. Two kinds of memory accesses are demonstrated for a comparison: 1) guest software writes a message to VMM region and 2) guest software writes a message to guest region. Third, provide a VM exit handler to handle the VM exit from the guest to see whether any exception happens.

The result of this experiment demonstrates that guest software can only access the guest region but can't access the VMM region. This is no surprise because of the appropriate EPT configuration illustrated in Section 12.3.2. Once the access is violated, an exception is caught by the VM exit handler.

#### 13.2.4.2 Interference between VMMs

In order to guarantee isolation, a program running inside a VMM_A must not interact with another program in VMM_B on the same machine. If a program on VMM_A is able to change memory or monitor VMM_B, then this is considered a serious isolation break. Section 13.2.4.1 demonstrated that guest software with low privilege level (Ring 3) cannot access to the host VMM with high privilege level (Ring 0), which conforms to the Bell-LaPadula (BLP) “No read up” security policy. One more question comes out, what if they are with the same privilege level? Are programs running on one VMM able to affect other VMMs running in the same privilege level? If such a potential security bug is exploited, an attacker can have access to all VMMs. That is why the memory management must be carefully configured and maintained.

We proposed a software mechanism (protection page table) for isolation to control the interference among all the VMMs. More details will be illustrated in Section 14.4 and Section 14.5.2.

### 13.2.5 System Registers Analysis

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the RFLAGS register and several system registers:

- The system flags and *IOPL* fields in the *RFLAGS* register control task and mode switching; interrupt handling, instruction tracing, and access rights.
- The control registers (*CR0*, *CR2*, *CR3*, *CR4*, and *CR8*[38]) contain a variety of flags and data fields for controlling system-level operations.
- The debug registers allow the setting of breakpoints for use in debugging programs and system software.
- The *GDTR*, *LDTR*, and *IDTR* registers contain the linear addresses and sizes of their respective tables.
- The task register contains the linear address and size of the *TSS* for the current task.
- The Model-Specific Registers (MSRs) control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type range registers (*MTRR*s).

IAVMM has the ability to extract system information and take snapshots so that a user can keep logs of a specific state for further analysis. Therefore, our VMM offers good transparency including presenting host and guest state information about Control Registers, MSRs, and so on. The following subsections describe some of these registers and demonstrate how they are implemented in our prototype.

#### 13.2.5.1 Control Registers

Control registers determine the operating mode of the processor and the characteristics of the currently executing task. In 64-bit mode, control registers are expanded to 64 bits. CR0 contains system control flags that control operating mode and states of the processor. CR3 contains the

[38] CR8 is available in 64-bit mode only, known as task priority register.

physical address of the base of the paging-structure hierarchy (e.g. the base address of the PML4 table), and two flags (PCD and PWT) which control caching of that paging structure in the processor's internal data caches. CR4 contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. Figure 46 depicts a picture of control registers in guest VM when a mini_guest VM is launched. According to Figure 47, the system enables paging (CR0.PG and CR0.PE) in guest VM. Since both the CR0.CD and CR0.NW flags are clear, caching of memory locations of the whole of physical memory in the processor's internal caches is enabled. IAVMM clears CR4.PSE flag to enable 2-MByte or 4-KByte pages without 4-MByte pages. CR4.PAE flag must be set to enable paging to produce physical addresses with more than 32 bits before entering IA-32e mode. Our system enables VMX operation in guest VM so the CR4.VMXE should be set.

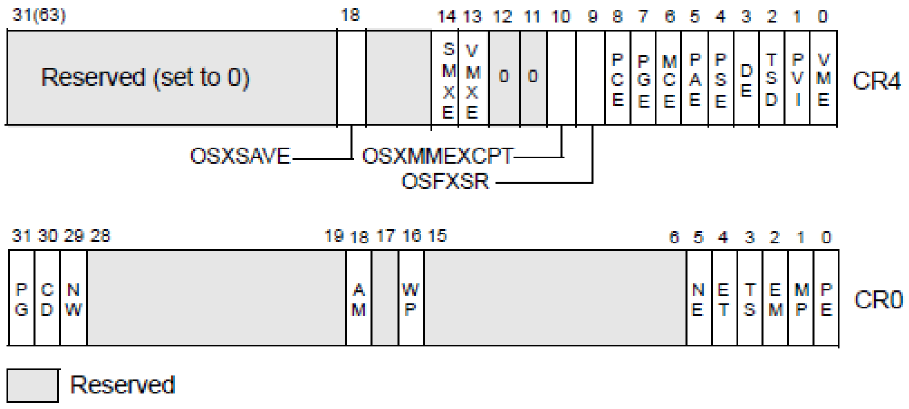


**Figure 47: The formats of CR0 and CR4**

### 13.2.6 VM Exits

VM exits occur in response to certain instructions and events (see Section 12.6) in VMX non-root operation. IAVMM provides VM Exit handler to deal with most of the VM exits. For example, Figure 46 shows a case that IAVMM captures VM exit information when a mini_guest VM is launched. In that case, “MWAIT exiting” is set in VM-Execution control fields during constructing VMCS. As the output shows, a VM exit occurs with a basic exit reason `0x24` when the instruction `VMLAUNCH` is executed to launch the guest virtual machine. Exit reason `0x024` indicates that an `MWAIT` instruction arrived and the “MWAIT exiting” VM-execution control was set.

In addition, IAVMM sets up the guest to cause a VM exit to the VMM on external interrupts. This is done by setting the “external-interrupt exiting” VM-execution control in the guest controlling-VMCS. Interrupts are automatically masked by hardware in the processor on VM exit by clearing *RFLAGS.IF*.

## 13.3 SUMMARY

Multicore processor architectures have been designed to provide efficient performance, but they introduce new security concerns, including shared memory issues, registers issues and so on. Virtualization brings a more complex and risky security environment. Most of security issues as well as VM escapes and VMBR have been discussed in section 10.4.2. One of our main goals is to evaluate hardware features in multicore processors, especially in the CBEA processor and Intel Core i7 processors, and then demonstrate how they have an impact on these multicore systems.

Through IAVMM, we are able to get a vision of processor information and system states, hence building a fundamental block to determine if there exist possible vulnerabilities in VM systems. To illustrate this, 26 sensitive but unprivileged instructions are all evaluated and the results show that most of them[39] are handled securely by the configuration of VMCS. Covert channels in multicore system are analyzed as well. The experiments about interference between multiple VM systems demonstrate that unauthorized subjects can be prevented from accessing protected resources, if the EPT is configured appropriately.

[39] Six of these instructions are not handled successfully in hardware for VM Exits.

# 14 A LAYERED FRAMEWORK FOR SECURE MULTICORE ARCHITECTURES

Assurance of multicore systems requires examination of all the hardware or software, to ensure that there are no accidental or malicious mechanisms that could allow malicious users or processes to reach the trusted region. Due to a wide variety of malicious behaviors, assurance would most likely have to specify exactly what each element of hardware and software is intended to perform, and to provide evidence that it does it correctly. Ideally we wish to specify a security framework for multicore architectures with any number of levels and components. However, for research convenience, we restrict ourselves to a 3-level system in this report. In this chapter, we analyze multicore architectures by examining each component from hardware level, up through hypervisor level until user level.

The remainder of this chapter is organized as follows: Section 14.1 gives some terms specifically used in this report. Section 14.2 introduces 3-level framework for secure multicore architectures. The following sections present the evaluation of each component within this framework. Section 14.6 is the summary of this chapter.

## 14.1 SECURITY POLICY TERMINOLOGY

The following are basic terms and concepts used in security policy research that are relevant to this project.

- **Security Policy**: A security policy describes the security requirements for a system. Security requirements are specific to a system and provide protection of essential services or resources.
- **Security Mechanism**: A security mechanism is a way of implementing security policies.
- **Security Model:** A security Model is a way of formalizing security policies.
- **Subject**: A subject is an entity in the computer system that performs operations.
- **Object**: An object is an entity in the computer system that is affected by the operations performed by subjects. An object may also be a subject.
- **Classification Levels**: Sensitive information is classified into different levels for which access is restricted by law or regulation to particular classes of people. A security clearance is required to handle classified documents or access classified data. There are typically several levels of sensitivity, with differing clearance requirements. This sort of hierarchical system of secrecy is used by virtually every national government[40]. For purposes of this report, the classification levels, as well as clearance levels, from the highest to the lowest are Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U).
- **Access Right**: An access right is the permission for a subject to access a particular object for a specific type of operation. A typical set of operations are *r,w,a,e*, where *r* represents

[40] Although we use military style levels, this can be mapped to other hierarchies as well.

that a subject can “read” an object, *w* represents that a subject can “write” and “read” an object, *a* represents that a subject can “write” (not “read”) an object, *e* represents that a subject can “execute” an object.

- **Confidentiality**: Confidentiality refers to limiting information access and disclosure to authorized users – “the right people” – and preventing access by or disclosure to unauthorized users – “the wrong people”. Breaches of Confidentiality can occur when data is not handled in a manner adequate to safeguard the confidentiality of the information concerned. In the literature, confidentiality is sometime referred to as secrecy or privacy.
- **Integrity**: Integrity refers to the trustworthiness of information resources. It includes the concept of “data integrity”, namely, that data has not been changed inappropriately, whether by accident or through deliberately malign activity. It also includes “origin” or “source integrity”, that is, the data actually came from an identified person or entity rather than an imposter.
- **Availability**: Availability refers to the availability of information resources. It is the assurance that the systems responsible for delivering, storing and processing information are accessible when needed. An information system that is not available when needed is at least as bad as none at all. It may be much worse, depending on how reliant the organization has become on a functioning computer and communications infrastructure.
- **Covert Channel**: A covert channel is a method of establishing illicit communication between subjects. Covert channels are realized through the use of either system variables as storage channels or forcing changes in the performance or response time of objects as timing channels.
- **Reference Monitor:** A reference monitor is a model of the portion of the computer system that checks all operations performed by a subject on an object to determine whether or not the operation is permitted by the security policy. The implementation of this monitor should be small enough to be verified, tamper-proof, unavoidable, and should check each operation performed.

## 14.2 3-LEVEL SECURITY FRAMEWORK

We analyze multicore architectures by examining each component at the hardware level, hypervisor level and user level. The conceptual framework of a 3-Level security policy architecture is depicted in Figure 48 and will be illustrated based on our IAVMM, which is developed for security analysis of Intel 64 architectural features. To be specific, we discuss this framework with 3 subsequent stages:

- Identify and examine multicore hardware architectural features;
- Decompose security policy in IAVMM level into pieces of components that can be mapped into hardware level;
- Verify that VMM- and Hardware- level security policy satisfies user-level security requirements.

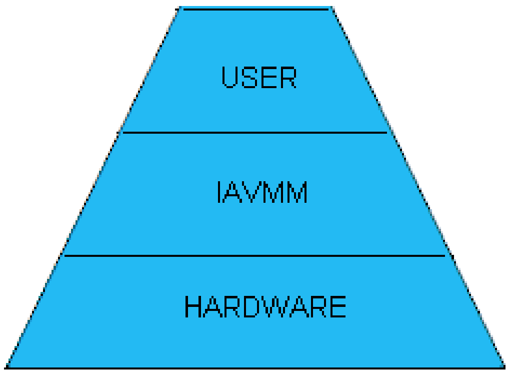


**Figure 48: 3-Level Security Policy Framework for Multicore Architectures**

We demonstrate an application system (see Figure 48) to illustrate how to accomplish these steps. The system is simple enough to avoid complexity for the purposes of discussion but sufficient to illustrate the concepts of our approach. Informally, it is a boxes-and-arrows diagram in which we want no channels for information flow other than those explicitly indicated by arrows. We present this architecture through the analysis of a two-partition system with an assumption that each partition system only contains one virtual machine system (VMS). For example, two generic secure building blocks, MMR and GC, are introduced in the VMM level to enforce security policies. MMR (Message Management Router) performs the partition identification, message labeling and routing of typed messages. The GC (Guard Controller) provides a guard for checking authorized message behavior. Specifically, MMR encodes messages with security levels, and GC enforces security policies.

The specifications of this system are (Figure 49(a)): two single-level users, one with top secret and the other one with secret[41], communicating with a multi-level database (DB). The database implementation is simple and assumes that messages from the users are tagged correctly with the correct security level.

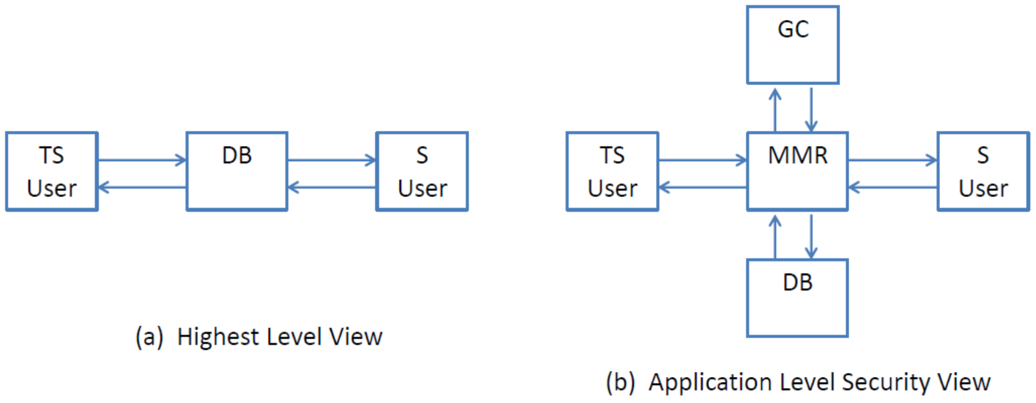


**Figure 49: Assurance architecture of the multicore system**

[41] In the Bell-LaPadula model of military-style classification, Four security levels are arranged from the most sensitive Top Secret, Secret, Confidential to the least sensitive Unclassified.

Assuming a Bell-LaPadula Model, which consists of the simple security property (no "read up") and the *-property (no "write down"), is applied at the user level, we need to make sure that the messages are labeled correctly so that the system will not violate the security property. The system designer can design an application level security system (Figure 49(b)) where messages from the users are sent through the MMR which first sends the messages to the GC which ensures that the messages are tagged correctly and then sends them to the DB. The MMR routes return messages to the correct users.

The system designer can take this implementation, place each component, the DB, MMR and GC into separate virtual machines and support the whole system on a multicore platform. The next step of our validation of this system then involves decomposing and mapping the high level policy (with knowledge of Secret and Top Secret users), into the VMM and eventually the hardware level of multicore architectures. To be specific, we analyze this mechanism based on memory paging system.

Moreover, even if there is an attempt to compromise the system, the policy architecture should be responsible for taking appropriate actions. It is possible that, with the exemplary 3-level security policy framework designed specifically for this implementation of Intel Core i7 processors, we could provide software solutions or safeguards for some of the security concerns found in this research. The results of our work could serve to guide future security development for the Intel Core i7 processors.

## 14.3 EXAMINATION OF HARDWARE-LEVEL SECURITY MECHANISMS

With recent advances in hardware virtualization support, hardware control mechanisms have the advantage of being self-constrained and have a smaller attack surface, thus reducing the chance of subversion. Consequently, we want to discuss the opportunity to consider how these mechanisms can increase the overall trustworthiness of virtualized systems. As for security mechanisms to be enforced yet in hardware level, we examine them, but are actually not limited to, from the categories below.

### 14.3.1 Secure Hardware Mechanisms of Intel VT-x

There are already many corresponding mechanisms enforced at the hardware level to provide protection according to security requirements.

- **Protection Rings**: The Virtual Machine Extension (VMX) in Intel VT-x is a new hardware enhancement introduced to support virtualization, where the privileged mode is labeled VMX root mode and the unprivileged mode is called VMX non-root mode. This VMX non-root mode can be regarded as a new mode of operation with reduced privileges. Both modes support all four rings, allowing the hypervisor to be able to use multiple privilege levels on its own and the guest OS can be allowed to execute at its intended ring level.

- **Instructions**: Before the advent of hardware virtualization extensions for Intel virtualization, 17 sensitive but unprivileged instructions violated the classical virtualization requirements presented in [PG74, RI00]. Currently with the support of Intel VT-x, these virtualization problems can be handled by the VMM. Since that time, Intel has added over 500 new instructions through various language extensions. We examined these additional instructions and discovered 9 more instructions that also fall into this category, including `MWAIT/MONITOR`, `XSAVE/XRSTOR`, `XSAVEOPT`, `RSM`, `XGETBV`, and `SYSCALL/SYSENTER`. These 26 sensitive but unprivileged instructions are listed in Appendix D. For example, the POPF instruction, one of sensitive and unprivileged instructions, pops a word from the top of the stack and stores the value in the lower 16 bits of the EFLAGS register, hence allowing values in the EFLAGS register to be changed that could impact the performance of another VM, thus violating classic virtualization concepts. Fortunately, Intel VT-x provides VM exits mechanisms to handle most of these instructions, if the VMM designer so chooses.

The behaviors of some instructions are changed in VMX non-root operation, since the way to execute these instructions is determined by the settings of certain VM-Execution control fields. For example, the execution of CPUID instruction in root operating mode retrieves the information of the processors, however, the execution of it in non-root operating mode causes a VM exit.

- **Memory Virtualization**: Memory Virtualization allows the VMM to enforce control of physical memory and yet support guest OSs' expectation to manage memory address translation. Intel VT-x includes two mechanisms for memory virtualization: Virtual Processor Identifier (VPID) and Extended Page Table (EPT). CPU caches memory translation information in the TLB, which is associated with a VM specific tag, VPID. This allows the TLB to keep track of which TLB entry belongs to which VM and the TLB entries of the different VMs can coexist peacefully in the TLB provided the TLB is big enough. When a logical processor performs a TLB invalidation operation, only the TLB entries that are tagged for that logical processor are guaranteed to be flushed. As a result, guest software can be allowed to handle its own page faults, thereby reducing the frequency of VM exits which avoids costly virtualization overhead.

  Rather than have the VMM manage the shadow page table's mapping, Intel's EPT adds a separate set of hardware-walked page tables which translate from guest physical addresses to host physical addresses that are used to access memory, which reduces context switches frequency within the guest thus alleviating hypervisor overheads.

- **Secure Hypervisor**: Establishing a secure hypervisor is essential and each VM will always behave in the expected manner and contain a minimum set of functions enabling a description of the platform characteristics. Intel Trusted Execution Technology (Intel TXT) provides hardware-based security technologies to build a solid foundation for security. Built into Intel's silicon, these technologies address the increasing and evolving security threats across physical and virtual infrastructure by complementing runtime protections such as anti-virus software. Intel TXT creates a Measured Launch Environment (MLE) that enables an accurate comparison of all the critical elements of the launch environment against a known good source. It creates a cryptographically unique identifier for each approved launch-enabled component, and then provides hardware-based enforcement mechanisms to block the launch of code that does not match approved code. This solution can protect against the software-based attacks that threaten integrity, confidentiality, reliability and availability of systems.

- **Secure I/O:** Intel VT-d adds an IOMMU to not only allow device memory addresses in DMA to be mapped to physical memory addresses automatically, but also provide a kind of memory protection. Without an IOMMU, a device can perform DMA to physical addresses that it should not be able to touch; with the IOMMU, such DMA requests can be blocked. The IOMMU can be configured such that a request from a particular device <bus #, device #, function # > can only have access to particular memory ranges, with any accesses outside those ranges being trapped as an error.

Based on the aforementioned analysis, Intel virtualization technology does provide hardware support features for security. As a result, systems that require a higher amount of security and assurance have traditionally turned to hardware-enforced security to gain an additional level of protection should the measures in the OS fail, such as recent Intel TXT and TPM functionality incorporated into commodity PCs. However, more and more covert channels are introduced with the Intel VT-x, i.e. the misconfiguration of Task Register would cause unexpected results. Highly secure systems may also require protection at VMM level to secure VM systems in multicore architecture.

## 14.4 VMM-LEVEL SECURITY MECHANISMS

This section presents the use of a protection page table to guarantee that the state of one VM system cannot affect other VM systems.

Even though Intel virtualization technology with hardware extensions only provides a coarse grained isolation scheme, IAVMM builds and implements a protection page table so that only subjects with authorized permissions can access a specific range of memory. Otherwise, the CPU will generate an exception on every attempt to access with unauthorized permissions. Once the hypervisor receives such an exception, it will stall the kernel if no exception handler deals with that exception. Table 52 presents access permissions in different VMX operation modes, wherein $R$, $W$ and $X$ stand for *read*, *write*, and *execute* permission, respectively. From the table, we configure the pages of VMM memory with either writable ($W$) or executable ($X$) in VMX root operation, but never both. This type of memory protection is generally referred to as $W \oplus X$ protection (exclusive or). VMM code is marked read-only for VMX non-root operation in the protection page table. This mechanism prevents any code executing in guest mode from modifying VMM code pages, thereby enhancing system robustness.

**Table 52: Protection page table enforced on IAVMM**

| | *VM Memory* | *VMM Data* | *VMM Code* |
|---|---|---|---|
| VMX Non-root Operation | RWX | W | - |
| VMX Root Operation | RWX | RW | RX |

Under this protection table scheme, the state of one VM system cannot be affected by the execution of other VM systems. In the case of two VM systems as implemented in IAVMM, by combining protection page table with the EPT built for resource isolation in Figure 40, we built an experiment to demonstrate whether VMM2 can access VMM1 successfully, discussed in Section 14.5.2.

## 14.5 VERIFICATION EXPERIMENTS FROM USER-LEVEL SECURITY MECHANISMS

This section lists the experiments used to verify if the IAVMM policy satisfies the user level security requirement in this IAVMM prototype. We will discuss two broad categories of the models of confidentiality: access control security and information flow security.

### 14.5.1 Access Control Security

Access control means that a guard controls whether a principal (the subject) is allowed access to a resource (the object). Suppose that we want to enforce policies of this kind in the multicore architecture. A typical example is the Bell-LaPadula (BLP) model. Two security rules should be imposed in the multicore architectures. First, VMM is allowed a read access to the data with a lower or equal security classification, such as resources in guest software. Second, Guest OS is allowed a write access to the data with an equal or higher security classification. The verification experiments are built in Section 13.2.2. Because of the appropriate configuration of EPT, the system can prevent guest software from accessing the host memory region.

### 14.5.2 Protection Page Tables

Combining protection page table with the EPT built for resource isolation in Figure 40, we built an experiment to demonstrate whether VMM2 can access VMM1 successfully. Whether the access is permitted by a translation or not is determined by access rights specified by those paging-structure entries which control the translations. The mechanism of access rights is performed with the following steps. In the host page tables for an application processor (e.g. an AP with APIC_ID #2), we disable *write* right by clearing the access flag *R/W* of each page-directory entry that maps a 2 MB page, hence protecting that region against writing. From the perspective of VMM2 as shown in Figure 40, we clear the access flag *R/W* in every paging-structure entry within the GVM1 and VMM1 region. In this way, the experiment result shows that the VMM2 can't access to the region of the VMM1. It will cause page-fault exception which makes VMM2 stall, and causes VMM1 to invoke a VM exit with exit reason 3, indicating that an *init* signal arrived.

However, another issue is uncovered with this experiment. A processor may cache information from the paging-structure entries in TLBs and paging-structure caches, and the processor may enforce access rights based on the TLBs and paging-structure caches instead of the paging structures in memory. If software modifies a paging-structure entry to change access rights, it must update the TLB to ensure that the processor uses the new change for a subsequent access to an affected linear address.

## 14.6 SUMMARY

In this chapter, we presented research results we have achieved based on the analysis of exemplary 3-level security framework for multicore architectures. Multicore processor architectures have been designed to provide efficient virtualization, but in the meantime they also introduce some security concerns. Two contributions in this chapter are to propose a layered framework to securely evaluate multicore architectures and to demonstrate how it has an impact on multicore systems involving verifications of hardware-level mechanisms, hypervisor-level and user-level mechanisms. The next chapter will present a way to enforce security policies for this framework.

# 15 FORMALIZE SECURITY POLICY FOR MULTICORE ARCHITECTURES

The design of a secure system requires architects to develop a system architecture that supports implementation of various security policies. Enforcing security policies at the architecture level is attractive because it allows security concerns to be recognized early and can be given sufficient attention in the design stages. A security policy, determined by regulation and doctrine, defines "secure" for a system. The term "security policy for multicore architectures" covers security requirements that protect multicore systems from any unauthorized behaviors. In short, to develop a secure multicore system, it is necessary to enforce a security policy framework as simple and as strong as possible to provide general guidance for secure system on multicore architectures. In this chapter, we introduce a layered assurance scheme for multicore architectures and illustrate (using 3 layers) how to formalize the framework. This chapter is an extension of work published earlier [AFHS11].

The remainder of this chapter is organized as follows: Section 15.1 gives a brief literature review of security properties. Section 15.2 introduces a formal model of virtual machine systems and brings out a security requirement for multicore architectures, followed by some example layered assurance. Section 15.3 is the summary of this chapter.

## 15.1 BACKGROUND

Virtualization technology for commodity processors has entered the hardware extensions era. Multicore architectures with hardware-assisted virtualization technology have become a prevalent platform for building virtual machine systems. One key characteristic of virtualization technology is isolation. The main principle of isolation is to guarantee that any application in one VM cannot affect other applications executing in a different VM, or that processes running in one VM cannot affect other VMs running in the same machine. If this security assumption is broken, then an attacker can have access to VMs in the same machine or even to the host system. In light of these considerations, a reasonable question is "how to protect VMs?" To protect against this potential vulnerability in high-assurance systems, there are two main steps that must be completed. The first is to obtain a thorough understanding of the desired security properties of a multicore system. This normally calls for a formal model of the system together with security proofs, often given in some mathematical or logical language. The second is to provide assurance that the implementation of a multicore system realizes the more abstract formal model. The research community has spent considerable effort on the first area, such as some work on security architecture modeling and on providing security architecture design guidance for architects [MQRG97, ZAF08, BDRS08]. However, there is still a lack of additional detailed guidance for multicore systems from the perspective of hardware level.

### 15.1.1 Layer Assurance Architectures

Computer security is concerned with system architectures that can allow unauthorized software to access sensitive information. A wide variety of layered assurance designs and verification for security architectures have evolved at the same time as both of them have a strong influence on each other. Most treatments of MILS [AFOTH06, WU05] have focused on a three-layer approach comprised of separation kernel, middleware, and applications, whereas Boettcher [BDRS08] and others characterize MILS by a two-level approach (policy level and resource sharing level) to secure information sharing. The policy level decomposes functional and security objectives for the system to yield a MILS policy architecture in which all security-critical functions are performed by trusted components. The trusted components enforce local security policies that work together to achieve the security policy of the overall system. Recently, University of Cambridge Computer Laboratory and SRI International's Computer Science Laboratory have been working on a hardware-software co-design approach [NW10] in which assurance is synergistic with the system architecture. This approach can be provided from the hardware up through the lower-layer software, in such a way that desired system security policy can be enforced architecturally to support a wide variety of software operations. This approach augments a current programmable hardware, Field Programmable Gate Array (FPGA) soft core, with new Capability Hardware Enhanced RISC Instructions (CHERI) features, which are designed to allow incremental adoption of higher-assurance approaches, with a focus on security-critical components making up contemporary TCBs: the separation kernel / hypervisor, OS kernel, and language runtimes. High assurance design may be supported at each layer in the system.

### 15.1.2 Security Architecture Models

We explore security architecture models by first defining the term security property. A security property is an instantiation of a security policy. There may be more than one property that satisfies a given policy. This section introduces a variety of security properties. A security framework generally assumes that there are at least two levels of users within a system, low-level (*L*) and high-level (*H*)[42] The intuitive notion is that high-level users should be protected from low-level users, and information should not flow from high-level users to low-level users. The purpose of a security property is to prevent low-level users from being able to make deductions about the events of the high-level users, to prevent unauthorized modification of data and to prevent users from affecting availability.

Many security properties are modeled on the concepts of event systems. An event system is a specification of the behavior of the system in terms of events (which could be mapped to state-transitions in a state-machine model). Associated with events are usually inputs and outputs of the system, where the outputs can be seen by users. A sequence of events is called a *trace*. The set of traces of the system is usually modeled as the set implemented by the system. For the implementation to satisfy the security property, we analyze the set of traces.

To understand security properties, we first need a more precise understanding of traces and specifically of Low Level Equivalency Sets (*LLES*). An *LLES* is a set of traces that have the same low-level events (in order) that a low-level user can observe. Formally, given a trace τ and a System *S*:

$$LLES(\tau, S) = \{s \mid \tau \mid L = s \mid L \wedge s \in traces(S)\} \qquad \text{Eqn. 3}$$

[42] This can be generalized into a lattice-based hierarchy, and does not necessarily imply military-style security levels.

The expression $\tau \mid L$ denotes the trace formed by removing from $\tau$ all events not in $L$. The function *traces*($S$) is the set of authorized traces in system $S$. The following security properties can be expressed with the *LLES* (Eqn. 3) notation [ZL97]:

- **Noninference [O'H90]:** A noninference security property requires that low-level users should not be able to infer information about high-level users.
- **Noninterference [GM84]:** A noninterference security property requires that high-level users are prevented from influencing the behavior of low-level users; otherwise, low-level users could infer information about high-level user activities.
- **Non-Deducible Output [GN88]:** A non-deducible output security property requires that low-level users cannot distinguish the events causing high-level users' output.
- **Separability [McL94]:** A separability security property requires that no interaction or information flow is allowed between low-level and high-level users.
- **Restrictiveness [McC87]:** A restrictiveness security property requires that changes in sequences of high level events do not affect future possible sequences of low level events.

A large body of evidence suggests that few designers ensure that their systems meet these properties. Besides, the separability security property is too strong because it doesn't allow any interaction between low-level users and high-level users. Consider a system where the only *Top Secret* (*TS*) level behavior is to echo all *Secret* (*S*) level output events to a *TS* level device for archiving. This system does not satisfy true separability. Zakinthinos [ZL97] presented the "perfect security property (PSP)", the weakest security property that does not allow information flow from high-level users to low-level users, but does allow high level outputs to be influenced by low level events.

### 15.1.3 Perfect Security Property

This section summarizes the details of the perfect security property. The PSP (Eqn 4)formulation is:

Eqn 4

$$\forall \tau \in traces(S) : \tau|L \in LLES(\tau,\ S) \wedge \forall p, s : p\hat{\ }s \in LLES(\tau,\ S)$$
$$\wedge\ s|H = () :$$
$$\forall \alpha \in H : p\hat{\ }(\alpha) \in traces(S) \Rightarrow p\hat{\ }(\alpha)\hat{\ }s \in$$
$$LLES(\tau,\ S)$$

In this formulation, the trace $p$ ^ $s$ refers to the trace formed by concatenating traces $p$ and $s$, the trace $s$ has no high-level events, while $p$ might have some high-level events. The possibility of $\alpha$ occurring between $p$ and $s$ is only dependent on the preceding high-level events in $p$. The idea behind PSP is the same as that behind Separability. All possible high-level activity and interleaving must be possible with all low-level activity. The difference is that PSP allows high-level outputs to be dependent on low-level events. This does not reduce security since the low-level user still will not know how he has influenced high-level outputs. One other benefit of PSP is that it is a composable property (as are separability and restrictiveness). If we decompose a system into individual components, and prove that each component satisfies PSP, then the composite system will satisfy PSP -- a property that is surprisingly not preserved by all security properties.

Unfortunately, with all of these security properties, analysts are often stuck with a single level of abstraction. Even with composability, we must use the same specification of events and security properties, no matter which level of abstraction of implementation we are evaluating. A lower level may not support concepts of security levels in the context of an application, but may support concepts of separation and controlled information flow. A good layering approach will allow us to transition between levels of abstraction mapping lower level security properties to higher level properties.

Real systems will also use multiple components, working together, to implement the system functionality and security policy. It is not necessarily true that each component will satisfy the full security property, but may satisfy a subset that when combined with other components are sufficient to satisfy the full system security property. A layered approach must also support the concept of sub-policies and partial implementation of security properties with composition.

## 15.2 FORMAL MODEL OF VIRTUAL MACHINE SYSTEMS

This section presents the formal models of multicore systems based on the analysis of a virtual machine system.

### 15.2.1 Formal Model of Virtual Machine Systems

Definition 1 presents the traditional model of a state machine. We can refine this model in many ways to represent a system that supports multiple virtual machines and multiple physical machines. The extensions require two things, first is a mapping from the simple model into a model that can represent multiple interacting state machines, one for each virtual machine, and second is a model that allows for the concurrent execution environment of a multicore processor.

An instantiation of the above model can present more details with respect to the contents of the state. Take a multicore system as an instance, to be concrete; a machine state is the concatenation of all possible entities' states, such as all CPU register values for each CPU, the entire contents of memory, the entire contents of stable storage, etc. Such architecture inherently gives the hypervisor a complete view of all system state information. Therefore, a simplified instantiation of a state machine can be comprised of a finite number of entities' states with each being represented by a tuple $< L, P, E, B >$. These four components of each entity state are privilege label $L$, instruction pointer $P$, entity $E$, and storage-bound $B$. Such a view provides the ability to thoroughly observe each specific state. Privilege Label $L$ is at least guest mode $g$ or host mode $h$. Actually Intel VT-x provides 8 protection rings (Host Rings 0-3 and Guest Rings 0-3), wherein Host/Guest Ring 1-2 are unused in most operating systems. The instruction pointer $P$ refers to an index into storage-bound $B$ if secure, indicating the next instruction to be executed. Entity $E$, also known as Object $O$, is comprised of CPU and I/O registers as well as volatile and stable system storage devices. The storage-bound part *B* gives an executable range of the memory for a specific VM. Suppose that the executable storage range *B* is of size *q* and an instruction produces the address *a*,

**Definition 1: The formal model of a state machine is**

- $M = (\Sigma, \sigma_0, T)$
- $\Sigma$ is the set of states of the system
- Initial State: $\sigma_0 \in \Sigma$
- $T : \Sigma \rightarrow \Sigma$ defines the allowed transitions between states.
- The notation $\sigma(p)$ denotes the substate of $\sigma$ that corresponds to the named resource, *p*, in the system.

We can find that the "memory trap" happens as a result of an attempt by an instruction to access an address which is outside of the bounds in B. If so, that means that the state of one VM might be affected by the execution of an instruction.

A transition *T* represents execution of an instruction or interaction with I/O with the ability to observe or change certain aspects of a system. The execution of an instruction *i* is a transition allowing the system state to switch from σ to σ'.

Definition 1 presents the traditional model of a state machine. We can refine this model in many ways to represent a system that supports multiple virtual machines and multiple physical machines. The extensions require two things, first is a mapping from the simple model into a model that can represent multiple interacting state machines, one for each virtual machine, and second is a model that allows for the concurrent execution environment of a multicore processor.

Definition 2 presents a refined model of a state machine, represented as a composite of multiple constituent state machines.

**Definition 2:**

The formal model of a state machine $M$ can be subdivided into a collection of composite state machines $< \sigma^i, \sigma_0^i, T^i >$ each representing a virtual machine of the system, or the hypervisor.

- $M = (M^1, M^2, \ldots, M^n)$ $n$-tuple representing the individual state machines in the composite machine, where $M^i = < \sigma^i, \sigma_0^i, T^i >$
- $\forall\, \sigma \in \Sigma$: $\sigma = cs(\sigma^1, \sigma^2, \ldots, \sigma^n)$ where $\sigma^1 \in \Sigma^i$
- Initial State: $\sigma_0 = cs(\sigma_0^1, \sigma_0^2, \ldots, \sigma_0^n)$,
- The notation $cs(s_1, \ldots, s_n)$ denotes the composite state of the system.
- The extraction function $\mathcal{S}^i(\sigma) = \sigma_i$ returns the portion of the composite state relevant to sub-machine $i$.
- $T(\sigma) = cs(\tau^1(S^1(\sigma)), \tau^2(S^2(\sigma)), \ldots, \tau^n(S^n(\sigma)))$ where $\tau^i \in T^i$

**State Machine Policy 1:** The intersection of substates must be restricted such that execution of $\tau^1$ does not interact with substate $\sigma^j$ in violation of the security property.

**State Machine Policy 2:** If the execution of $\tau^i$ as part of $\tau \in T$ modifies a component of substate $\sigma^j$ ($j \neq i$), then the transition $\tau^j$ in $\tau$ must also specify that modification.

In this model, the full machine state is represented as a composite state $cs(s_1, \ldots, s_n)$. This is not an $n$-tuple, but the result of a composition function $cs$. This allows us to model systems where the constituent state machines share some portion of the overall machine state. Although we allow for shared state, we model system transition as a composite of individual transitions of the component state machines. This model allows for execution of multiple sub-machines simultaneously (modeling multicore and hyper-threading processing).

Note, the model allows too much interaction between sub-machines, hence the inclusion of the two *State Machine Policies*.

- *State Machine Policy 1 (SMP1)* is included to enforce the idea that we must only allow for authorized interactions between the sub-machines. If the overall policy permits interaction, then the sub-machines can share state, otherwise they should not.
- *State Machine Policy 2 (SMP2)* exists for consistency in the models. If sub-machine $i$ can cause a change in state for sub-machine $j$, then that possible change of state should be

modeled as a possible transition in sub-machine *j*. This allows for a cleaner independent analysis of each machine, and then analysis of their composition.

The next thing we have to worry about in this model is abstraction layering. Given a model of a state machine *M* at a low level in the system model (say a model of the hypervisor), how do we map that into the supporting infrastructure requirements of the higher level virtual machines? We support this using standard abstraction function, $\mathcal{A}$, that maps lower level states into higher level abstractions. The mapping of transitions will be managed through our event model, discussed below.

We say a portion of a state, $x \in \sigma$, is *hidden* in the abstraction if for two equivalent higher level abstract states $\sigma_1 = \sigma_2$, there exist different lower level states $\exists\ \sigma'_1 : \mathcal{A}(\sigma'_1) = \sigma_1$ and $\exists\ \sigma'_2 : \mathcal{A}(\sigma'_2) = \sigma_2$ where *x* has different values, $x_1 \in \sigma'_1 \neq x_2 \in \sigma'_2$. Note that hiding portions of the state may require that the abstract state-machine model become non-deterministic. Although this is okay, some security properties evaluated at the abstract level may fail when the non-determinism is removed through refinement to the lower-level implementation.

For this model to work correctly, the security property (and associated mapping functions) have to ensure that sub-machines *i* cannot modify a hidden portion of the state that would impact the behavior of sub-machine *j*.

### 15.2.2 Event System Model

We now define a formal model of events in our system, and a mapping between these events and the state transitions defined in the preceding section. Definition 3 defines an event as an action, *a*, acting on behalf of a subject, *s*, using data from a set of resources (objects), *r* and possibly modifying members of a set of resources, *w*. This generic representation of event can be used at any layer of our architecture, with an understanding that a representation at one layer will be an abstraction of the representation at the next lower layer. Our state-machine model presented in the previous section, defines transitions as atomic changes to the state of a system. Events are more complex actions that may involve multiple interactions with the state, and are therefore represented/implemented by a sequence of state transitions as shown in Definition 3[43]. Events that are specified as atomic actions do not allow for interference during their sequence of state transitions, whereas synchronizing events do allow interaction with other synchronizing events. We will address this point later in the chapter.

In Definition 3 we introduce two *Event Policies*. These policies place restrictions on the semantics of the events, specifically on the mapping of events to a sequence of transitions and the use of objects.

- *Event Policy 1 (EP1)* specifies that the sequence of transitions may not modify any object that is not specifically listed in the *write-set*.
- *Event Policy 2 (EP2)* is a specification of determinism and completeness, which states that for two different states of the system, if the contents of objects in the *read-sets* of the

[43] We will need to check if (*a*,*s*,*r*,*w*) is permitted for a particular state of the system.

event are the same, then the contents of all objects in the *write-set* will be the same after the event.

**Definition 3: The formal model for events *E* are defined as follows:**

- $E = \{(a, s, r, w) \mid a \in A,\ s \in S,\ r, w \in P(O)\}$ is the set of events of the system.
- $S$ is the set of subjects
- $O$ is the set of objects and $P(O)$ is powerset (set of subsets) of $O$.
- $r$ and $w$ are two (not-necessarily disjoint) subsets of objects that are accessed by action $a$; the *read-set* and the *write-set*.
- $E$ is the set of events, where events correspond to state transition chains let $\tau = \tau_0, \tau_1, \ldots, \tau_n \in T^*$ be a sequence of state transitions corresponding to event
  $e = (a, s, r, w)$ such that:

  $$\tau(x) = \tau_n(\tau_{n-1}(\ldots \tau_1(\tau_0(x)) \ldots))$$

  let $\sigma$ be the state of the system prior to execution of event $e$ and $\sigma^I = \tau(\sigma)$ be the state after execution of $\tau$.

- Events are classified as *atomic* or *synchronizing*

**Event Policy 1:** $\forall o \in O : o \notin w \Rightarrow \sigma^I(o) = \sigma(o)$

**Event Policy 2:** $\forall \sigma_1, \sigma_2 \in \Sigma : (\forall o \in r \cup w : \sigma_1(o) = \sigma_2(o)) \Rightarrow (\forall p \in w : \sigma^I(p) = \sigma^I(p))$

Let us consider the ramifications of these policies. First, since the event is specified with respect to an abstraction of the system state (sets of objects), it is possible that a portion of the system state that is not part of a specified object could be modified by the event. *EP1* does not prohibit this. For example, the instruction pointer and system clock registers are part of the system state, but may not be relevant in a discussion of events at a higher level of abstraction. However, even with the incompleteness of this policy, *EP2* ensures that the contents of these non-modeled state components cannot interfere with the computation of the event. For example, if an event wishes to copy the instruction pointer into a data structure, *EP2* requires that the instruction pointer be specified as an object in the *read-set* of the event. Otherwise, there may be two states of the system that differ only in the program counter and would therefore satisfy the antecedent of the implication but not satisfy the consequent, violating the policy.

A layered approach to using the event model can be readily developed using the standard commuting theories, as depicted in Figure 50 and Definition 4. We implement higher level events through sequences of lower level events, defined by the implementation function $\mathcal{I}$. We can abstract the lower level state to the higher lever state with the abstraction function $\mathcal{A}$. We then compose the system into events from multiple components at the same level using Definition 5.

**Definition 4: The formal model for commuting:**

- $\mathcal{I}$ is an implementation function (mapping event $e$ at a high level to a sequence of events $e_1, e_2, \ldots$ at a lower level.
- $\mathcal{A}$ is an abstraction function that maps the state of the lower level to the higher level of abstraction
- If $I(e) = e'_1 , e'_2 , e'_3$ ; $\tau$ is the transition sequence for $e$ and $\tau'_1$,‘$\tau'_2$, $\tau'_3$ are the transition sequences for $e'_1$, $e'_2$, $e'_3$, respectively, then these functions commute if each of the following is true

  - $\tau(\sigma_1) = \sigma_2 \ \wedge\ (\tau'_3 \circ \tau'_2 \circ \tau'_1)(\sigma'_1) = \sigma'_2$
  - $\mathcal{A}(\sigma'_1) = \sigma_1 \ \wedge\ \mathcal{A}(\sigma'_2) = \sigma_2$

**Definition 5: The formal model for events in a composite system are:**

- $\mathrm{E} \subseteq P(E)$. At any time there may be any number of “active” events in the system.

- Let $e \in \mathrm{E}$ be a set of active events, and $e_i$, $e_j \in e$ be two different active events.

  – If $e_i$ and $e_j$ are atomic events, then
  $$(e_i.r \cap e_j.w) = (e_i.w \cap e_j.w) = (e_i.w \cap e_j.r) = \emptyset$$
  – If $e_i$ is a synchronizing event, and
  $$(e_i.r \cap e_j.w) \neq \emptyset \ \vee\ (e_i.w \cap e_j.w) \neq \emptyset \ \vee\ (e_i.w \cap e_j.r) \neq \emptyset$$
  then $e_j$ and $e_i$ must be *partner synchronizing events*.

$$\begin{tikzcd}
\sigma_1 \arrow[rrr, "e"] & & & \sigma_2 \\
\sigma'_1 \arrow[u, "\mathcal{A}"] \arrow[r, "e'_1"] & {} \arrow[r, "e'_2"] & {} \arrow[r, "e'_3"] & \sigma'_2 \arrow[u, "\mathcal{A}"']
\end{tikzcd}$$

**Figure 50: Standard Commuting Diagram for Implementation Correctness**

Although this sounds straight forward, there are many issues that must be addressed to ensure that this abstraction does not violate the security properties of the system. These issues include:

- The State Machine and Event Policies defined earlier must hold.
- There must be a clear abstraction mapping between objects of the two levels (referenced in the read-sets and write-sets of events at both levels).
- For all *hidden* components, *x*, of a state, the value of the hidden component does not change the abstraction or interpretation functions, or cause a violation of the security property or correctness of the commuting diagram.

Now, with the model in place, we can define security properties, such as non-interference, or PSP at the top level of abstraction. For that property to hold, the issues mentioned above must be satisfied. However, it is not required that the lower level have the same "view" of the world; just that it support the higher level view. This is the normal model for computing languages; high level abstractions are refined to lower level implementations, and will work for security as well.

Now that we have an intuitive understanding of theoretical model how to represent a multicore system and then how to improve its security, how can we implement separate secure of each VM in current multicore architectures? We will illustrate the security requirements with an exemplary 3-layer framework of the machine model in the following subsection.

## 15.3 EXAMPLE LAYERED ASSURANCE

Consider a system such as that depicted in Figure 48-Figure 49 and in Figure 51. In Figure 51 (a) we have the hierarchy of authorized states of the system. The hardware supports a large set of authorized system states (states that will not cause a hardware failure or exception). However, we have the ability to configure the hardware (the CFG box in Figure 51 (b)) to limit these states. This configuration consists of setting execution modes and programming MMU page tables, VM modes, etc. The VMM configures the hardware to a limited subset of the allowable states, and the user can configure the VMM.

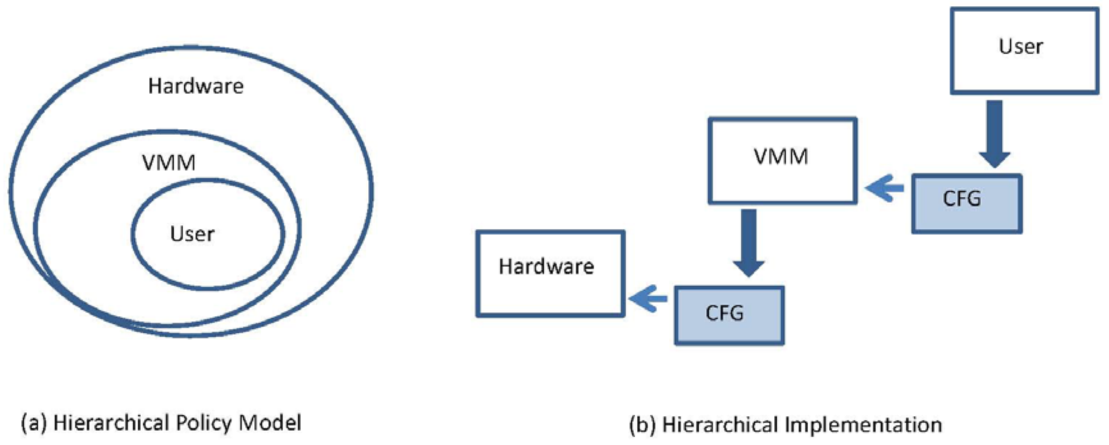

Hardware
VMM
User
(a) Hierarchical Policy Model
User
VMM
CFG
Hardware
CFG
(b) Hierarchical Implementation

**Figure 51: Layered Policy and Implementation Models**

At the highest layer of abstraction, we have the user view of the world, as depicted in Figure 49(a) and the system designer view (Figure 51(b)). At the highest level, we have an understanding of the security policy, we have subjects and objects mapped to security levels (e.g., Top-Secret and Secret). At the system designer view we understand the security properties of the system. The system designer defines executable entities such as users mapped with security levels, communication channels, and supporting hypervisor level resources such as guards and Extended Page Tables (EPT); all in support of the policy. The system implementer needs to map events such as "VM1 writes message to a specific location inside VM2" into a sequence of events which include locating the memory address of the message by traversing the EPT of the VMS2[44], verifying the memory writeability for VM1, and storing the message into the location VM1 specifies.

We must prove that the security property enforced at this level supports the high level security policy. We do that with many assumptions about the underlying infrastructure and the security policy exported by the lower level. Here we can possibly use PSP and the LLES definitions, but may need to add concepts such as non-transitive information flow to allow trusted implementations of the hypervisor and corresponding EPT.

### 15.3.1 Hardware Layer

At the lowest level we have the hardware platform which supports the concepts of execution in a context. A context is defined as the execution mode (e.g., supervisor/user, privilege ring, VM status) and the set of available resources (e.g., the memory maps in the MMU). The hardware layer provides mechanisms to set and change the configurations of contexts, and to perform context changes.

Subjects in the hardware model are mapped to the contexts that the hardware supports. All events are bound to the current executing subject of the hardware. In a multicore model, there may be multiple subjects, one running on each logical processor, or on a collection of processors. The subjects of the hardware are the physical resources of the hardware, memory, devices, registers, the MMU, etc. In the end, the hardware exports a model of an executing set of systems, the individual logical processors, and current executing contexts.

The security policy of the hardware does not directly map to the concepts of high and low-level users. However, we can still map the security policies of the hardware to allowable sequences of events $e_1^{\mathrm{hw}}, e_2^{\mathrm{hw}}, \ldots, e_n^{\mathrm{hw}}$. These will be of the form $\overline{e_1^{\mathrm{hw}}}, \overline{e_2^{\mathrm{hw}}}, \ldots \overline{e_n^{\mathrm{hw}}}$ where each $\overline{e_i^{\mathrm{hw}}}$ will be sequence of events for current contexts, or the context switch events. The set of traces for the hardware, *traces*(*HW*), will only contain those event that are allowable within the current contexts. If the context supports virtual memory and MMU-based memory maps, the events will corresponding to available virtual memory accesses and MMU maps. At this level of abstraction, verification of the correct behavior of the system includes verification that the hardware supports the configuration data and does not violate the contexts. The security property of the hardware must clearly specify the limitations of the hardware execution model and

[44] VMS2 refers to Virtual Machine System 2, which contains not only VM2, but also VMM2 system.

configuration. We should not attempt to model security levels, or user intent at this level, just the correct implementation of the configuration data.

### 15.3.2 VMM Layer

The VMM layer is responsible for defining the contexts of the hardware, and thus will only authorize a subset of the allowable states with a more restrictive policy (Figure 51). The hardware provides the basic security mechanisms of isolation: *virtualization*, *virtual memory*, *memory-management*, and *context switching*. It also supports multiple execution units. The hardware supports execution of each logical CPU core in either root-mode or in a guest (virtual machine) mode. The VMM configures the memory maps, assigns usage of the cores to the VMs and manages transitions in and out of virtualization mode.

The events of the VMM now correspond to actions of the individual VMs (in the environment of the VMM) and the control events of the VM (configuring the hardware, establishing the VM contexts). The VMM model of the system partitions the physical resources of the hardware into the subset of resources available to the individual VMs. Events at the VMM level will mostly still be at the same granularity of the hardware level, with additions for VMM specific actions (e.g., creation of a VM, swapping in/out a VM). However, the subjects in the VMM are now mapped to individual VMs and the VMM itself. The objects of the system are still mostly the hardware resources, but also now the VMM data structures representing the contexts of the VMs, possible buffers and other internal resources. We need a mapping of these objects onto the Hardware and a mapping of the VMM-specific events into Hardware events.

In the end, the VMM exports a model of an executing set of virtual systems, the individual VMs, and current executing contexts for those VMs. We still cannot model security levels, but we have now separated the hardware (time and space) into VM contexts, and can have VMM rules for behavior of those VMs. For example, if the VMM provides some simple services for inter-VM communications, or for VM managements, the VMM configuration data must indicate permissions for access to those services. The user will utilize the services of the VMM through configuration data, and verification of the VMM assures that it supports the policy specified by the configuration data (e.g., no communication occurs between VMs unless authorized in the VMM configuration data).

### 15.3.3 User Layer

At the highest layer of abstraction, we have the user view of the world, as depicted in Figure 48(a) and the system designer view (Figure 48(b)). The VMM does not have a concept of top secret or secret, routers, databases or guards. The VMM has the concepts of virtual machines, shared memory regions, inter-VM communications, and private resources. The VMM also maps and schedules VMs to particular cores. The configuration of the VMM is what enables us to provide the abstraction to the user level. The user level designer can utilize the VMM concepts of separation and controlled information flow to design a system similar to Figure 48(b). This system will implement guards and routers to ensure properly labeled and filtered messages between two users (with guards, routers and users all mapped to individual virtual machines).

The system designer defines executable entities such as users mapped with security levels, communication channels, and supporting application level resources such as guards and routers;

all in support of the policy. The system implementer needs to map events such as “send message to DB” into a sequence of events which include sending message to MMR, routing from MMR to GC, tagging the message in GC, sending it back to MMR, and then routing it from MMR to DB. We must prove that the security property enforced at this level supports the policy. We do that with many assumptions about the underlying infrastructure and the security policy exported by the lower level. Here we can possibly use PSP and the LLES definitions, but may need to add concepts such as non-transitive information flow to allow trusted implementations of the hypervisor and corresponding EPT.

## 15.3.4 More Specific Examples

Let's dive into the details of the above example, which will specify how to decompose the user level events into a sequence set of hardware level events. The example will show how events proceed when the system performs “VM1 writes message to a specific location inside VM2” and then a VM exit occurs. First, locate the memory address of the message by traversing EPT. In this event ($a'_1$,$s'_1$,$r'_1$,$w'_1$) from a hardware point of view $a'_1$ refers to *locate* memory address, $s'_1$ is the subject user in VM1, and host physical memory address is retrieved through the traversal of EPT in VMS2, which means that $r'_1$ is the EPT. The next event ($a'_2$,$s'_2$,$r'_2$,$w'_2$) will examine whether this range of memory can be accessed by VM1? The translation from guest physical addresses to physical addresses is determined by a set of EPT paging structures. EPT also provides the privileges to decide if access is allowed or not, such as read access (bit 0), write access (bit 1) and execute access (bit 2) in each EPT PML4 entry, EPT PDPTE, EPT PDE, and EPT Page Table Entry (the details of these entries are illustrated in Section 0). If an EPT page structure entry contains an unsupported value in the course of translation the memory address of the message, a EPT violation will occur due to such an attempt and cause a VM Exit. In this event, $a'_2$ refers to *read* access, and $r'_2$ is the EPT. The third event ($a'_3$,$s'_3$,$r'_3$,$w'_3$) due to the VM exit, records information about the nature and reason for the VM exit in the VM-exit information fields. $a'_3$ refers to *write* access, and $w'_3$ includes *exit_qualification* register, *guest_linear_address* field and *guest_physical_address* field in the corresponding VM structure. The final event will be the trap to the VMM. According to Figure 50, Figure 52 shows how events proceed when a VM exit happens.

$$\sigma_1' \xrightarrow{(\alpha_1', s_1', r_1', w_1')} \xrightarrow{(\alpha_2', s_2', r_2', w_2')} \xrightarrow{(\alpha_3', s_3', r_3', w_3')} \sigma_2'$$

$$\text{wherein } (\alpha_1', s_1', r_1', w_1') = (\textit{locate}, \text{user, EPT, EPT}),$$

$$(\alpha_2', s_2', r_2', w_2') = (\textit{read}, \text{system, EPT, EPT}),$$

$$(\alpha_3', s_3', r_3', w_3') = (\textit{write}, \text{system, registers, registers})$$

**Figure 52: Event Procedures of a VM Exit Example**

Another example covers instruction execution, which might cause VM exits unconditionally or based on the settings of VM-execution controls. Let's check one specific example, and illustrate how to decompose the user level event into a sequence set of hardware level events. The example demonstrates when guest software attempts to execute PAUSE and the “PAUSE exiting" VM-execution control was 1. First, user attempt to execute PAUSE instruction. In this event $(a'_1,s'_1,r'_1,w'_1)$ from a hardware perspective, $a'_1$ refers to *execute*, $s'_3$ is the subject thread in VM1. The next event $(a'_2,s'_2,r'_2,w'_2)$ will examine VMM level to check if the "PAUSE exiting" VM execution control is set? If not, the instruction executes normally. Otherwise, such an attempt causes a VM Exit. In this event, $a'_2$ refers to *read*, and $r'_2$ is the VM execution control field. Once a VM exit happens, the third event $(a'_3,s'_3,r'_3,w'_3)$ records information about the nature and reason for the VM exit in the VM-exit information fields. $a'_3$ refers to *write*, and $w'_3$ includes *VM-exit instruction-length* register, and *VM-exit instruction information* register in the corresponding VM structure. The final event will be the trap to the VMM.

## 15.4 SUMMARY

Intel and AMD have offered hardware-assisted virtualization technology to enable efficient virtualization in their CPU design. These advances have created a window of opportunities to provide a profound impact on the reliability of multicore systems, however, a high assurance scheme must be proven to correctly implement the security functions in its specifications and effectively mitigate risks to a level commensurate with the value of the assets it protects [LIN06]. Therefore, the design of a secure multicore system requires architects to develop a system architecture that satisfies necessary security policies for this multicore system. One important contribution of our work is the modeling of a layered bottom-up security policy framework in terms of multiple secure architectures that are related by formal mappings. We have proposed an approach for specifying and verifying a layered assurance scheme (from hardware layer, through hypervisor layer, up to user layer) for multicore architectures.

# 16 PART III CONCLUSION AND FUTURE WORK

Multicore processors are becoming ubiquitous in the enterprise because they can be utilized to get higher performance and efficiency by splitting system tasks into subtasks and distributing them across the multicore chipset. As one of the most important multicore architectural features, virtualization technology has spurred a number of exciting research with virtual machines, especially in the area of hardware-assisted extensions. Both Intel and AMD have introduced hardware virtualization extensions in their current mainstream processors, such as Intel VT-x and AMD-V. However, while providing services in a virtual machine gains benefits, it has also led to explosive growth of security concerns in virtual machine systems.

## 16.1 CONCLUSION

To evaluate security of virtualization technologies in multicore architectures, this report first examined hardware features for virtualization technology in multicore processors. Furthermore, by taking advantage of hardware virtualization support, we developed a lightweight virtual machine monitor prototype, called IAVMM, which is purpose-built for Intel 64 architecture and security analysis of multicore systems. In order to strengthen reliability in multicore systems, we proposed a modeling of a layered bottom-up security policy framework in terms of multiple secure architectures that are related by formal mappings.

### 16.1.1 Summary of Part III of this project

To evaluate the virtualization technologies in multicore systems, we first examined the CBEA architecture. Our paper [SHAF10b] presents an overview of the CBEA processor and potential security concerns in this architecture, such as security vulnerabilities of local storage and registers for each SPU. An experimental result also demonstrated that the “unfair” memory access mechanism does not exist in CBEA architecture. Second, we evaluated hardware-assisted virtualization technologies in multicore processors, such as Intel VT-x and AMD-V. They provide faster performance for virtualization and simplify VMM implementations. Intel VT-x extensions are conceptually the same as those implemented in AMD-V, including CPU virtualization, memory virtualization and I/O virtualization.

To develop a platform that allows us to run a VM system and evaluate the security features of the underlying hardware features, we decided to build our own lightweight VMM, called IAVMM, based on Intel 64 architecture [HAF11]. By taking advantage of hardware virtualization support and concentrating only on security analysis functionality issues, we were able to keep it thin and simple. IAVMM is capable of extracting BIOS and system information and offering good transparency to present host and guest states, hence building a fundamental block to determine if there exist possible vulnerabilities in VM systems. Through IAVMM, 26 sensitive but unprivileged instructions were all evaluated and the results show that 20 of them are handled securely via the proper configuration of VMCS. Programmed exit conditions in IAVMM also help us to invoke a VM exit for convenience of experimental security analysis. In order to comprehensively examine hardware features and security concerns in multicore architectures, we wake up multiple logical processors in this platform and performed experiments on security analysis. The experiments about interference between multiple VM systems demonstrate that unauthorized subjects can be prevented from accessing protected resources, if the EPT is configured appropriately. Our IAVMM is really a lightweight VMM compared with the commodity hypervisors. Even though it doesn't provide full functionalities of commodity VMMs, it is enough for research experimentations and extensible if more features are needed.

As the demand for system virtualization grows, so does the need to handle the security challenges existing in VM systems, such as VM escapes and VMBRs. The design of a secure system requires architects to develop a system architecture that satisfies security policies. In order to improve our understanding of security in multicore systems, we present and examine a layered framework (from hardware level, through hypervisor level, up to user level) for secure multicore architectures, and introduce a layered assurance scheme for such architectures [AFHS11]. We proposed an approach for specifying and verifying this layered assurance scheme for multicore architectures.

The work presented in this report leads to a better understanding of virtualization technologies and security concerns in multicore architectures, which will benefit architects and researchers by providing security evaluation of hardware features and a lightweight virtual machine monitor for security analysis of such architectures. Moreover, a layered assurance scheme is provided in this report to assist in the evaluation of security for multicore architectures design.

## 16.2 FUTURE WORK

The previous sections have summarized the achievements of this report. However, as in every research effort, the results have indicated a number of issues where further work lead to other important conclusions. Some of these areas include:

1. **More Experiments**: Now that we have a platform (IAVMM) for evaluating the security of virtualization technologies in multicore architectures, specifically in Intel 64 architecture. We presented a lot of security concerns based on this platform from the hardware viewpoint. However, research efforts are still worthy of digging out more potential security issues as well as more experiments.

2. **IAVMM platform for AMD**: IAVMM can only support the Intel 64 architecture and can be used as a platform for security analysis and experimentation of the hardware-based virtualization technologies. However, AMD has also released AMD-V. Therefore, IAVMM should support AMD64 to be used for security analysis in AMD64 architecture as well in the future.

3. **Nested virtualization**: The idea of nested virtualization [BYDD$^+$10] is to run multiple other virtual machines in a virtual machine. Of course the concept can be extended to recursive virtualization for the x86 architecture. As commodity operating systems gain virtualization functionality, nested virtualization will be required to run those operating systems in virtual machines. For example, Windows 7 already supports a compatible Windows XP mode; that is, Windows XP runs in a virtual machine.

Unfortunately, in Intel VT-x, executing VMLAUNCH instruction in a VM environment (VMM non-root operating mode) causes a VM exit, which would prevent the nested virtualization from being implemented in the Intel 64 architecture. There are also other compelling concerns about nested virtualization, which are as follows:

a. Nested VMX virtualization for nested CPU virtualization;
b. Multi-dimensional paging for nested MMU virtualization;
c. Multi-level device assignment for nested I/O virtualization.

4. **Enforcement of security policy**: Research efforts are still underway to verify the security requirements in our security policy framework, and provide the safeguards to mitigate the risks. Even if the system has been compromised, the policy architecture should be responsible for taking appropriate action.

In summary, evaluation of multicore architectures is an area full of exciting research, and we will continue our efforts to examine virtualization technology in multicore architectures in order to demonstrate how this key hardware component has impact on the construction of secure multicore systems.

# Part IV

# REFERENCES AND APPENDICES

# 17 BIBLIOGRAPHY

[AA00] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *In International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2000.

[AFHS11] Jim Alves-Foss, Xiaohui He, and Jia Song. Layered assurance scheme for multicore architectures. In Layered Assurance Workshop, 2011.

[AFOTH06] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3/4):239–247, 2006.

[AJM+06] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and JohnWeigert. Intel Virtualization Technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, August 2006.

[AMD09] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*, Feb 2009.

[BDF+03] aul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[BDRS08] Carolyn Boettcher, Rance DeLong, John Rushby, andWilmar Sifre. The MILS component integration approach to secure information sharing. In *IEEE/AIAA Digital Avionics Systems Conference*, pages 1.C.2 (1–14), 2008.

[BL76] D.E. Bell and L.J. LaPadula. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA: HQ Electronic System Division, March 1976.

[BLRS08] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and SeanW. Smith. Traps, events, emulation, and enforcement: managing the yin and yang of virtualizationbased security. In *ACM Workshop on Virtual Machine Security*, pages 49–58, 2008.

[BOAFS10] R. Bradetich, P. Oman, J. Alves-Foss, and J. Smith. Towards resilient multicore architectures for real-time controls. In *International Symposium on Resilient Control Systems*, 2010.

[Bri] Peter Bright. The Ars technical guide to I/O virtualization.
http://arstechnica.com/business/guides/2010/02/io-virtualization.ars/
Last accessed Nov, 2010.

[BYDD+10] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Cha10] D. Champagne. *Scalable Security Architecture for Trusted Software*. PhD thesis, Princeton University, 2010.

[CN01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *HotOS*, pages 133–138, 2001.

[DEG] L. DuFlot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. fawlty.cs.usfca.edu/˜cruse/cs630f06/duflot.pdf; accessed 31-Mar-2012.

[Dep85a] U.S. Department of Defense, editor. *Guidance for Applying the Department of Defense Trusted Computer Systems Evaluation Criteria in Specific Environments* ("*Light Yellow Book*"). Fort George G. Meade, Maryland 20755-6000, June 1985.

[Dep85b] U.S. Department of Defense, editor. *A Guide to Understanding Trusted Recovery* ("*Yellow Book*"). Fort George G. Meade, Maryland 20755-6000, June 1985.

[Dep87] U.S. Department of Defense, editor. *A Guide to Understanding Discretionary Access Control in Trusted Systems* ("*Orange Book*"). Fort George G. Meade, Maryland 20755-6000, Sept. 1987.

[Dep93] US Department of Defense. *A Guide to Understanding Covert Channel Analysis of Trusted Systems* ("*Light Pink Book*"). Fort George G. Meade, Maryland, 1993.

[Dep01] U.S. Department of Commerce. *Security Requirements for Cryptographic Modules*, 2001.

[DLM+06] Yaozu Dong, Shaofan Li, Asit Mallick, Jun Nakajim, Kun Tian, Xuefei Xu, Fred Yang, and Wilfred Yu. Extending Xen with Intel virtualization technology. *Intel Technology Journal*, 10(3):193–203, August 2006.

[Dou10] H. Douglas. Thin hypervisor-based security architectures for embedded platforms. Master's thesis, The Royal Institue of Technology, Stockholm, Sweden, 2010.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[ELND09] Eric Lacombe, Cincent Nicomette, and Yves Deswarte. Enforcing kernel constraints by hardware-assisted virtualization. *Journal in computer virology*, August 2009.

[FAD+06] B. Flachs, S. Asano, S. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra and J. Liberty, B. Michael, H. Oh, S. Mueller, I. Takahashi, A. Hatakeyama, Y. Wantanaby, N. Yano and D. Brokenshire, M. Peyravian, V. To, and E. Iwata. The microarchitecture of the synergistic procoprocessor for a cell processor. *IEEE Journal of Solid-State Circuits*, 41(1), 2006.

[FO06] John Fisher-Ogden. Hardware support for efficient virtualization. http://cseweb.ucsd.edu/˜jfisherogden/hardwareVirt.pdf, 2006.

[Fra06] M. Franz. Moving trust out of application programs: A software architecture based on multi-level security virtual machines. Technical Report TR. 06-10, 2006.

[Fre11a] Freescale Semiconductor. *e500mc Core Reference Manual*, 2011. Doc. #E500MCRM, Rev.0.

[Fre11b] Freescale Semiconductor. *EREF 2.0: A Programmer's Reference Manual for Freescale Power Architecture Processors*, 2011. Doc #EREF RM, rev.0.

[Fre11c] Freescale Semiconductor. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, 2011. Doc. #P4080RM, rev.0.

[Fre11d] Freescale Semiconductor. *QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual*, 2011.

[Fre12] Freescale Semiconductor. *P4080 Rev. 2 Security (SEC 4.0) Reference Manual*, 2012. Doc. #P4080SECRM, rev.2.

[GM84] Joseph A. Goguen and Jos´e Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[GN88] Joshua D. Guttman and Mark E. Nadel. What needs securing. In *CSFW*, pages 34–57, 1988.

[Gol72] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Symposium on Network and Distributed Systems Security*, 2003.

[GRU] Grub legacy. http://www.gnu.org/software/grub/grub-legacy.en.html.

[HAF11] Xiaohui He and Jim Alves-Foss. A lightweight virtual machine monitor for security analysis on Intel 64 architecture. *Journal of Computing Sciences in Colleges*, 25(1), 2011.

[HHOAF05] W. S. Harrison, N. Hanebutte, P. Oman, and J. Alves-Foss. The MILS architecture for a secure global information grid. *Crosstalk: The Journal of Defense Software Engineering*, 18(10):20–24, Oct 2005.

[HOL+05] N. Hanebutte, O. Oman, P.M. Loosbrock, A. Holland, W.S. Harrison, and J. Alves-Foss. Software mediators for transparent channel control in unbounded environments. In *Proc. Information Assurance Workshop*, pages 201 – 206, June 2005.

[Hyp] Hyper-v project. http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx/.

[IBM] IBM Corporation. Cell Broadband Engine Architecture. http://www.01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/CBEA_v1.02_11Oct2007_pub.pdf.

[IBM07a] IBM Corporation. *Cell Broadband Engine CMOS SOI 90 nm Hardware Initialization Guide* v1.5, 2007.

[IBM07b] IBM Corporation. *Cell Broadband Engine Programming Handbook*, April 2007.

[ine] P4 series – p4080 multicore processor. http://cache.freescale.com/files/netcomm/doc/fact_sheet/QorIQ_P4080.pdf?fpsp=1; accessed 19-Mar-2011.

[ine07] Tilera now shipping the TILE64 processor: the world's highest performance embedded processor. http://www.marketwire.com/press-release/tilera-now-shipping-tile64-processor-worlds-highest{-performance-embedded-processor-}761947.htm accessed 19-mar-2011. 2007.

[ine09] Key architectural features AMD Athlon$^{TM}$x2 dual-core processors. 2009.

[ine11] Moores law and Intel innovation. http://www.intel.com/about/companyinfo/museum/exhibits/moore.htm, accessed 23-Mar-2011.

[Int08] Intel Corporation. *Intel Virtualization Technology for Directed I/O Architecture Specification*, Sep 2008.

[Int09a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Dec 2009.

[Int09b] Intel Corporation. Intel trusted execution technology (Intel TXT) software development guide. Technical Report Document Number: 315168-006, Intel Corporation, December 2009.

[Int12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual:* 1, 2A, 2B, 2C, 3A, 3B, and 3C, 2012.

[JWX10] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through VMM-base "out-of-the-box" semantic view reconstruction. *ACM Trans. Information System Security*, 13(2), 2010.

[Kan06] K. Kaneda. Tiny virtual machine monitor, Jun 2006. http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/

[Kan08] David Kanter. Inside Nehalem: Intel's future processor and system. http://realworldtech.com/page.cfm?ArticleID=RWT040208182719&p=2; Last accessed Feb, 2010.

[Kar05] Paul Karger. Multi-level secure requirements for hypervisors. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 267–276, 2005.

[KCW+06] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[LIN06] Timothy E. Levin, Cynthia E. Irvine, and Thuy D. Nguyen. Least privilege in separation kernels. In *International Conference on Security and Cryptography*, pages 355–362, 2006.

[Mal08] D. Maliniak. Software rules the day in multicore SoC design. http://electronicdesign.com/article/eda/software-rules-the-day-in-multicore-soc-design1864.aspx, 2008.

[MBH+07] D. Marr, F. Binns, D. Hill, G. Hilton, D. Koufaty, J. Miller, and M. Upton. Hyperthreading technology architecture and microarchitecture. *Intel Technical Journal*, 6(1), Feb. 2007.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, pages 161–166, 1987.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. *In IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[MM07] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multicore systems, 2007.

[MQRG97] Mark Moriconi, Xiaolei Qian, Robert A. Riemenschneider, and Li Gong. Secure software architectures. In *IEEE Symposium on Security and Privacy*, pages 84–93, 1997.

[MS00] Robert Meushaw and Donald Simard. NetTop — commercial technology in high assurance applications. *Tech Trend Notes*, 9(4), 2000.

[NALS06] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *MICRO*, pages 208–222, 2006.

[NSJ+09] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. MAVMM: Lightweight and purpose built VMM for malware analysis. In *ACSAC*, pages 441–450, 2009.

[NSL+06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[NVD] http://nvd.nist.gov/. Last accessed Nov 1, 2010.

[NW10] Peter Neumann and Robert Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Layered Assurance Workshop LAW*, 2010.

[O'H90] C. O'Halloran. A calculus of information flow. In Acte de ESORICS 90, Toulouse, pages 147–159, October 1990.

[pap09] An introduction to the Intel quickpath interconnect. 2009.

[PCIa] Conventional PCI. http://www.pcisig.com/specifications/conventional/.

[PCIb] PCI Express. http://www.pcisig.com/specifications/pciexpress/.

[Per05] C. Percival. Cache missing for fun and profit. 2005.

[PG74] Gerald J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7), 1974.

[PPPC07] L. Peng, J. Peir, T. Prakash, and Y. Chen. Memory performance and scalability of Intel's and AMD's dual-core processores: A case study. In *Proc. of the IEEE International Conference on Performance, Computing and Communications*, 2007.

[PSE09] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Workshop on Virtual Machine Security*, pages 1–10, November 2009.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson, and John D. Owens. Memory access scheduling. In *ISCA,* pages 128–138, 2000.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *USENIX Security Symposium*, pages 129–144, 2000.

[ROAF+06] B. Rossebo, P. Oman, J. Alves-Foss, R. Blue, and P. Jaszkowiak. Using spark-ada to model and verify a mils message router. In *Proc. International Symposium on Secure Software Engineering*, Mar. 2006.

[Rob99] J. Robin. *Analyzing the Intel Pentium's capability to support a secure virtual*. Master's thesis, Naval Postgraduate School, 1999.

[RT07] Joanna Rutkowska and Alexander Tereshkin. Is game over(), anyone? Technical report, Caesars Palace, Las Vegas, August 2007.

[Rus81] J. Rushby. Design and verification of secure systems. *ACM Symposium on Operating System Principles*, 15(5):12–21, December 1981.

[Rut08] Joanna Rutkowska. Security challenges in virtualized environments. In *RSA Conference*, 2008.

[RWW] M. Riley, J.Warnock, and D.Wendel. Cell broadband engine processor: Design and implementation. http://www.csd.uoc.gr/˜hy529/docs/riley.pdf; Accessed: 09/25/2011.

[SAF06a] J. Son and J. Alves-Foss. Covert timing channel analysis of rate monotonic realtime scheduling algorithm in MLS systems. In *Proc. IEEE Information Assurance Workshop*, June 2006.

[SAF06b] J. Son and J. Alves-Foss. Covert timing channel capacity of rate monotonic real-time scheduling algorithm in MLS systems. In *Proc. IASTED International Conf. on Communication, Network and Information Security*, pages 13–19, Oct. 2006.

[SAF07] J. Son and J. Alves-Foss. High level specification of non-interference security policies in partitioned MLS systems. In *Proc. IASTED International Conf. on Communication, Network and Information Security*, Sept. 2007.

[SAF09] J. Son and J. Alves-Foss. A formal framework for real-time information flow analysis. *Computers & Security*, (6):421–432, 2009.

[SC06] United States and William L. Clay. *Information assurance [electronic resource] : national partnership offers benefits, but faces considerable challenges : report to the Honorable William Lacy Clay, House of Representatives*. U.S. Government Accountability Office, [Washington, D.C.] :, 2006.

[Sca08] M. Scarpino. *Programming the Cell Processor For Games, Graphics, and Computation*. Prentice Hall, Ann Arbor, MI, 2008.

[SET+09] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takahi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kato, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: A thin hypervisor for enforcing I/O security. In *ACM SIGPLAN & ACM SIGOPS Conf. on Virtual Execution Environments*, pages 121–130, 2009.

[SHAF10a] Jessica Smith, Xiaohui He, and Jim Alves-Foss. A security review of the Cell Broadband Engine processor. In *Hawaii International Conference on System Sciences*, 2010.

[SHAF10b] Jessica Smith, Xiaohui He, and Jim Alves-Foss. A security review of the cell broadband engine processor. In *HICSS*, pages 1–8, 2010.

[SK09] H. P. Hofstee S. Keckler, K. Olukotun. *Multicore Processors and Systems*. Springer, 2009.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *Operating Systems Review*, 41(6):335–350, December 2007.

[Smi10] J. Smith. *In investigation of hardware security in multicore architectures*. Master's thesis, University of Idaho, 2010.

[SN05] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32 – 38, May 2005.

[Son06] Sony Corporation. *Cell Broadband Engine Architecture* v1.01, 2006.

[Suh05] G. E. Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, 2005.

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[TGC87] C.-R. Tsai, V. D. Gligor, and C. S. Chandersekaran. A formal method for the identification of covert storage channels in source code. *In IEEE Symposium on Security and Privacy*, pages 46–48, 1987.

[U.S93] U.S. Department of Defense. *A guide to understanding covert channel analysis of trusted system.* ("Light Pink Book") NCSC-TG-030, National Computer Security Center, November 1993.

[Vir] http://www.virtualbox.org/

[VMwa] http://www.vmware.com/esx/

[VMwb] http://www.vmware.com/products/workstation/

[VMWc] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4496. Last accessed Jan 31, 2011.

[VMw07] VMware Inc. *Understanding full virtualization, paravirtualization, and hardware assis*t. Technical report, November 2007.

[WJ10] Zhi Wang and Xuxian Jiang. HyperSafe: *A lightweight approach to provide lifetime hypervisor control-flow integrit*y. Report, Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 2010.

[WRa] R. Wojtczu and J. Rutkowska. Attacking Intel trusted execution technology. http://invisiblethingslab.com/resources/bh09dc/AttackingIntelTXT-paper.pdf; accessed 31-Mar-2012.

[WRb] R. Wojtczu and J. Rutkowska. Attacking Intel TXT via SINIT code execution hijacking. http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf; accessed 31-Mar-2012.

[WRc] R. Wojtczu and J. Rutkowska. Attacking SMM memory via Intel CPU cache poisoning. http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf; accessed 31-Mar-2012.

[WRd] R. Wojtczu and J. Rutkowska. Following the white rabbit: Software attacks against Intel VT-d technology. http://www.invisiblethingslab.com/resources/2011/SoftwareAttacksonIntelVT-d.pdf ; accessed 31-Mar-2012.

[WRT] R.Wojtczu, J. Rutkowska, and A. Tereshkin. Another way to circumvent Intel trusted execution technology. http://invisiblethingslab.com/resources/misc09/AnotherTXTAttack.pdf; accessed 31-Mar-2012.

[WU05] W. Mark Vanfleet, R. William Beckwith, Dr. Ben Calloni, Jahn A. Luke, Dr. Carol Taylor and Gordon Uchenick. MILS:architecture for high assurance embedded computing. *CrossTalk - The Journal of Defense Software Engineering*, August 2005.

[Xena] http://www.xen.org/

[Xenb] http://bugzilla.xensource.com/bugzilla/show_bug.cgi?id=1068 Last accessed Jan 31, 2011.

[ZAF08] Jie Zhou and Jim Alves-Foss. Security policy refinement and enforcement for the design of multi-level secure systems. *Journal of Computer Security*, 16(2):107–131, 2008.

[ZL97] Zakinthinos and Lee. A general theory of security properties. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

[ZLZZ08] Hongzhong Zheng, Jiang Lin, Zhao Zhang, and Zhichun Zhu. Memory access scheduling schemes for systems with multicore processors. In *ICPP*, pages 406–413, 2008.

# 18 ACRONYM LIST

| | |
|---|---|
| ACPI | Advance Configuration and Power Interface |
| AMD | Advanced Micro Devices |
| AMP | Asymmetric MultiProcessor |
| ARM | Advanced RISC Machines |
| AP | Application Processors |
| APIC | Advanced Programmable Interrupt Controllers |
| ASID | Address Space IDentifier |
| ATO | Atomic Unit |
| BE | Broadband Engine |
| BEI | Boadband Engine Interface Unit |
| BED | Cell BE Distribution |
| BIC | Bus Interface Controller |
| BIU | Bus Interface Unit |
| BLP | Bell-LaPadula |
| BU | Branch Unit |
| BUI | Bus Interface Unit |
| BM | Buffer Manager |
| BSP | BootStrap Processor |
| C | Confidential |
| CBEA | Cell Broadband Engine Architecture |
| CCB | CHA Cluster Block |
| CCSR | Configuration, Control, and Status Register |
| CD | Cache Disabled |
| CFX | Complex Integer Instruction |
| CHA | Cryptographic Hardware Accelerators |
| CHERI | Capability Hardware Enhanced RISC Instructions |
| CIU | Core Interface Unit |
| CMOS | Complementary Metal-Oxide-Semiconductor |
| CPL | Current Privileged Level |
| CPU | Central Processing Unit |
| CR | Control Register |
| CR3 | Control Register 3 |
| CSDS | Center for Secure and Dependable Systems |
| DCSR | Debug Control and Status Registers |
| DDR | Double Data Rate |
| DECAR | DECrementer Auto-Reload Register |
| DECO | DEscriptor Controllers |
| DEV | Device Exclusion Vector |
| DMA | Direct Memory Access |
| DMAC | Direct Memory Access Controller |
| DoD | Department of Defense |
| DoS | Denial of Service |

| | |
|---|---|
| DPAA | Data Path Acceleration Architecture |
| DPL | Descriptor Privilege Level |
| DRAM | Dynamic Random Access Memory |
| DS | Data address Space |
| DTLB | Data Translation Lookaside Buffer |
| DXE | Data Examination Engine |
| EA | Effective Address |
| EAL | Evaluated Assurance Level |
| ECC | Error Checking and Correction |
| EE | External Enabled |
| EIB | Element Interconnect Bus |
| eLBC | enhanced Local Bus Controller |
| EP | Event Policy |
| EPCR | Embedded Processor Control Register |
| EPT | Extended Page Table |
| EPTP | EPT base Pointer |
| EPU | Event Processing Unit |
| ESX | Elastic Sky X |
| FE | Floating-point Exception |
| FIR | Fault Isolation Registers |
| FlexIO | Frambus Flexible I/O |
| FM | Frame Manager |
| FPGA | Field Programmable Gate Array |
| FPSCR | Floating-Point Status and Control Register |
| FPU | Floating-Point Unit |
| FSB | Front-Side Bus |
| FCM | Flash Control Machine |
| GAO | Government Accountability Office |
| GART | Graphics Aperture Remapping Table |
| GCS | Guarded Communication Subsystem |
| GDT | Global Descriptor Table |
| GDTR | Global Descriptor Table Register |
| GP | General Purpose |
| GPe | General protection exeception |
| GPCM | General Purpose Chip select Machine |
| GPIO | General Purpose Input/Output |
| GS | Guest State |
| GVM | Guest VM |
| gPT | guest Page Tables |
| HAMES | High Assurance Middleware for Embedded Systems |
| HT | Hyper-Threading |
| HVM | Hardware-assisted Virtual Machine |
| HW | Hardware |
| IAVMM | Lightweight VMM |
| IC | Integrated Circuit |
| ICR | Interrupt Command Register |

| | |
|---|---|
| IDS | Intrusion Detection Systems |
| IDT | Interrupt Descriptor Table |
| IDTR | Interrupt Descriptor Table Register |
| IF | Interrupt Flag |
| IIC | Internal Interrupt Controller |
| IO | Input/Output |
| IOC | Input/Output Controller |
| IOIF | Input/Output InterFace |
| IOMMU | Input/Output Memory Management Unit |
| IOPT | Input/Output Page Table |
| IOST | Input/Output Segment Table |
| IO Trans | Input/Output address Translation |
| IPI | InterProcessor Interrupts |
| IS | Instruction address Space |
| IT | Information Technology |
| IVOR | Interrupt Vector Offset Register |
| JDKEK | Job Descriptor Key Encryption Key |
| JTAG | Joint Test Action Group |
| KES | Key Element Scanner |
| LA | Logical Address |
| LAW | Local Access Window |
| LDT | Local Descriptor Table |
| LDTR | Local Descriptor Table Register |
| LIODN | Logical I/O Device Number |
| LLES | Low Level Equivalency Sets |
| LRU | Least Recently Used |
| LS | Local Store |
| LSU | Load/Store Unit |
| L1 | Level 1 cache |
| L1CSR2 | L1 Cache Control and Status Register 2 |
| L1D | L1 Data cache |
| L1I | L1 Instruction cache |
| L2 | Level 2 cache |
| L2CSR1 | L2 Cache Control and Status Register 1 |
| MAVMM | Malware Analysis Virtual Machine Monitor |
| MDR | Message Data Register |
| ME | Machine check Enable |
| MFC | Memory Flow Controller |
| MIC | Memory Interface Controller |
| MILS | Multiple Independent Levels of Security |
| MLE | Measured Launch Environment |
| MLS | Multi-Level Secure |
| MLS-VM | Multi-Level Security Virtual Machine |
| MMIO | Memory  Mapped Input/Output |
| MMR | MILS Message Router |
| MMU | Memory Management Unit |

| | |
|---|---|
| MPH | Memory Performance Hog |
| MPIC | Multicore Programmable Interrupt Controller |
| MSI | Message Signaled Interrupts |
| MSR | Machine State Register |
| MSRs | Model-Specific Registers |
| MTRR | Memory Type Range Registers |
| NCU | NonCacheable Unit |
| NEAT | Non-bypassable, Evaluatable, Always invoked and Tamperproof |
| NAL | Nexus Aurora Link |
| NMI | Non-Maskable Interrupt |
| NOC | Network-on-a-Chip |
| NPC | Nexus Port Controller |
| NPT | Nested Page Tables |
| NSA/NCSC | National Security Agency/National Computer Security Center |
| NXC | Nexus Concentrator |
| OCN | On-Chip Network |
| OEM | Original Equipment Manufacturer |
| OS | Operating System |
| PAMU | Peripheral Access Management Units |
| PAT | Page Attribute Table |
| PC | Program Counter |
| PCI | Peripheral Component Interconnect |
| PCIe | Peripheral Component Interconnect express |
| PCS | Partitioned Communication Service |
| PME | Pattern Matching Engine |
| PMH | Page Miss Handler |
| PMFA | Pattern Matcher Frame Agent |
| PML | Page Map Table |
| POR | Power-On-Reset |
| PPC | Power PC |
| PPE | PowerPC Processor Element |
| PPU | PowerPC Processor Unit |
| PPSS | PowerPC Processor Storage Subsystem |
| PRV | PerVasive Logic |
| PS | Page Size |
| PXU | Processor execution Units (without L1) |
| QM | Queue Manager |
| RA | Real Address |
| RAS | Reliability, Availability, Serviceability |
| RISC | Reduced Instruction Set Computing |
| RMT | Replacement management Table |
| RMI | RapidIO Message Unit |
| RTIC | Real Time Integrity Checker |
| S | Secret |
| SCN | SPU Control |
| SD/MMC | Secure Digital / MultiMedia Cards |

| | |
|---|---|
| SEE | Security Encryption Engine |
| SFP | SPU Floating-Point |
| SFS | SPU Odd Fixed Point |
| SFX | SPU Even Fixed-Point |
| SIMD | Single Instruction, Multiple Data |
| SIP | System-in-a-Package |
| SIPI | Startup InterProcessor Interrupt |
| SKINIT | Secure Kernel Init |
| SLS | SPU Load and Store |
| SMI | System Management Interrupt |
| SMM | System Management Mode |
| SMT | Simultaneous Multi-Threading |
| SOC | System-on-a-Chip |
| SPE | Synergistic Processing Elements |
| SPI | Serial Peripheral Interface |
| SPM | State Machine Policy |
| SPU | Synergistic Processing Unit |
| SRAM | Static Random-Access Memory |
| SRE | Stateful Rule Engine |
| sRIO | serial RapidIO |
| SRM | Shared Resource Matrix |
| STM | SMM Transfer Monitor |
| SSC | SPU Channel |
| SVM | Secure Virtual Machine |
| SXU | Synergistic Execution Unit |
| TCB | Trusted Computing Base |
| TCSEC | Trusted Computer Security Evaluation Criteria |
| TCU | Test Control Unit |
| TDKEK | Trusted Descriptor Key Encryption Key |
| TDSK | Trusted Descriptor Signing Key |
| TKM | Token management Unit |
| TLA | Trace Logic Array |
| TLB | Translation Look aside Buffers |
| TS | Top Secret |
| TSS | Task-State Segment |
| TPM | Trusted Platform  Module |
| TVMM | Tiny Virtual Machine Monitor |
| U | Unclassified |
| UART | Universal Asynchronous Receiver/Transmitter |
| UNCLE | User-mode Cache Lock Enabled |
| UPM | User-Programmable Machines |
| USB | Universal Serial Bus |
| VA | Virtual Address |
| VLIW | Very Long Instruction Word |
| VM | Virtual Machine |
| VMBR | VM-Based Rootkit |

| | |
|---|---|
| VMCS | Virtual-Machine Control data Structure |
| VMI | Virtual Machine Introspection |
| VMM | Virtual Machine Monitor |
| VMS | Virtual Machine Systems |
| VMX | Virtual Machine Extensions |
| VPID | Virtual Processor ID |
| VPN | Virtual Private Network |
| VT-x | Virtualization Technology |
| XDR | Extreme Data Rate |
| XIO | XDR I/O |
| XCR | Extended Control Register |
| 10GE | 10- Gigabit Ethernet Controller |

# Appendix A. CBEA SPPORT DATA

## A.1 SPE MEMORY LOAD/STORE INSTRUCTIONS

**Table 53: SPE Memory Load/Store Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Load Quad Word | lqd | Yes | 1.0 |
| Load Quad Word | lqx | Yes | 1.0 |
| Load Quad Word | lqa | Yes | 1.0 |
| Load Quad Word Relative | lqr | Yes | 1.0 |
| Store Quad Word | stqd | Yes | 1.0 |
| Store Quad Word | stqx | Yes | 1.0 |
| Store Quad Word | stqa | Yes | 1.0 |
| Store Quad Word Relative | stqr | Yes | 1.0 |
| Generate Controls for Byte Insertion | cbd | Yes | 1.0 |
| Generate Controls for Byte Insertion | cbx | Yes | 1.0 |
| Generate Controls for Halfword Insertion | chd | Yes | 1.0 |
| Generate Controls for Halfword Insertion | chx | Yes | 1.0 |
| Generate Controls for Word Insertion | cwd | Yes | 1.0 |
| Generate Controls for Word Insertion | cwx | Yes | 1.0 |
| Generate Controls for Doubleword Insertion | cdd | Yes | 1.0 |
| Generate Controls for Doubleword Insertion | cdx | Yes | 1.0 |

## A.2 SPE CHANNELS

**Table 54: SPE Channels**

| CH | Constant | R/W | Blk | Cap. | Purpose |
|---|---|---|---|---|---|
| 0 | SPU_RdEventStat | R | B | 1 | Read SPU Event Status |
| 1 | SPU_WrEventMask | W | N | 1 | Write Event Mask |
| 2 | SPU_WrEventAck | W | N | 1 | Write Event Acknowledge |
| 3 | SPU_RdSigNotify1 | R | B | 1 | Read Signal Notification 1 |
| 4 | SPU_RdSigNotify2 | R | B | 1 | Read Signal Notification 2 |
| 5 | | | | | |
| 6 | | | | | |
| 7 | SPU_WrDec | W | N | 1 | Write to SPU Decrementer |
| 8 | SPU_RdDec | R | N | 1 | Read SPU Decrementer |
| 9 | SPU_WrMSSyncReq | W | B | 1 | Write to MS Synchronization Register |
| 10 | | | | | |
| 11 | SPU_RdEventMask | R | N | 1 | Read SPU Event Mask |
| 12 | SPU_RdTagMask | R | N | 1 | Read SPU Tag Mask |
| 13 | SPU_RdMachStat | R | N | 1 | Read SPU Machine Status |
| 14 | SPU_WrSRR0 | W | N | 1 | Write to Save/Restore Register |
| 15 | SPU_RdSRR0 | R | N | 1 | Read from Save/Restore Register |
| 16 | MFC_LSA | W | N | 1 | MFC Local Storage Address |
| 17 | MFC_EAH | W | N | 1 | MFC Effective Address High |
| 18 | MFC_EAL | W | N | 1 | MFC Effective Address Low |
| 19 | MFC_Size | W | N | 1 | MFC Transfer/List Size |
| 20 | MFC_TagID | W | N | 1 | MFC Command Tag ID |
| 21 | MFC_Cmd | W | B | 16 | MFC Class ID |
| 22 | MFC_WrTagMask | W | N | 1 | Write to MFC Tag Group Mask |
| 23 | MFC_WrTagUpdate | W | B | 1 | Write to MFC Tag Update Request |
| 24 | MFC_RdTagStat | R | B | 1 | Read MFC Tag Group Status |
| 25 | MFC_RdListStallStat | R | B | 1 | Read Stall-and-Notify Tag |
| 26 | MFC_WrListStallAck | W | N | 1 | Write to Stall-and-Notify Ack |
| 27 | MFC_RdAtomicStat | R | B | 1 | Read Atomic Command Status |
| 28 | SPU_WrOutMbox | W | B | 1 | Write to SPU Outbound Mailbox |
| 29 | SPU_RdInMbox | R | B | 4 | Read from SPU Inbound Mailbox |
| 30 | SPU_WrOutIntrMbox | W | B | 1 | Write to SPU Outbound Interrupt |
| 31 | | | | | |

## A.3 SPE CONSTANT-FORMATION INSTRUCTIONS

**Table 55: SPE Constant-Formation Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Immediate Load Halfword | ilh | Yes | 1.0 |
| Immediate Load Halfword Upper | ilhu | Yes | 1.0 |
| Immediate Load Word | il | Yes | 1.0 |
| Immediate Load Address | ila | Yes | 1.0 |
| Immediate Or Halfword Lower | iohl | Yes | 1.0 |
| Form Select Mask for Bytes Immediate | fsmbi | Yes | 1.0 |
| Add Halfword | ah | Yes | 1.0 |

## A.4 SPE INTEGER AND LOGICAL INSTRUCTIONS

**Table 56: SPE Integer and Logical Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Add Halfword | ah | Yes | 1.0 |
| Add Halfword Immediate | ahi | Yes | 1.0 |
| Add Word | aw | Yes | 1.0 |
| Add Word Immediate | awi | Yes | 1.0 |
| Subtract from Halfword | sfh | Yes | 1.0 |
| Subtract from Halfword Immediate | sfhi | Yes | 1.0 |
| Subtract from Word | sf | Yes | 1.0 |
| Subtract from Word Immediate | sfi | Yes | 1.0 |
| Add Extended | addx | Yes | 1.0 |
| Carry Generate | cg | Yes | 1.0 |
| Carry Generate Extended | cgx | Yes | 1.0 |
| Subtract from Extended | sfx | Yes | 1.0 |
| Borrow Generate | bg | Yes | 1.0 |
| Borrow Generate Extended | bgx | Yes | 1.0 |
| Multiply | mpy | Yes | 1.0 |
| Multiply Unsigned | mpyu | Yes | 1.0 |
| Multiply Immediate | mpyi | Yes | 1.0 |
| Multiply Unsigned Immediate | mpyui | Yes | 1.0 |
| Multiply and Add | mpya | Yes | 1.0 |
| Multiply High | mpyh | Yes | 1.0 |
| Multiply and Shift Right | mpys | Yes | 1.0 |
| Multiply High High | mpyhh | Yes | 1.0 |
| Multiply High High and Add | myphha | Yes | 1.0 |
| Multiply High High Unsigned | mpyhhu | Yes | 1.0 |
| Multiply High High Unsigned | mpyhhau | Yes | 1.0 |

*Continued on next page*

*Table 56– continued from previous page*

| | | | |
|---|---|---|---|
| Count Leading Zeros | clz | Yes | 1.0 |
| Count Ones in Bytes | cntb | Yes | 1.0 |
| Form Select Mask for Bytes | fsmb | Yes | 1.0 |
| Form Select Mask for Halfwords | fsmh | Yes | 1.0 |
| Form Select Mask for Words | fsm | Yes | 1.0 |
| Gather Bits from Bytes | gbb | Yes | 1.0 |
| Gather Bits from Halfwords | gbh | Yes | 1.0 |
| Gather Bits from Words | gb | Yes | 1.0 |
| Average Bytes | avgb | Yes | 1.0 |
| Absolute Differences of Bytes | absdb | Yes | 1.0 |
| Sum Bytes into Halfwords | sumb | Yes | 1.0 |
| Extend Sign Byte to Halfword | xsbh | Yes | 1.0 |
| Extend Sign Halfword to Word | xsbw | Yes | 1.0 |
| Extend Sign Word to Doubleword | xsbd | Yes | 1.0 |
| And | and | Yes | 1.0 |
| And with Complement | andc | Yes | 1.0 |
| And Byte Immediate | andbi | Yes | 1.0 |
| And Halfword Immediate | andhi | Yes | 1.0 |
| And Word Immediate | andi | Yes | 1.0 |
| Or | or | Yes | 1.0 |
| Or with Complement | orc | Yes | 1.0 |
| Or Byte Immediate | orbi | Yes | 1.0 |
| Or Halfword Immediate | orhi | Yes | 1.0 |
| Or Word Immediate | ori | Yes | 1.0 |
| Or Across | orx | Yes | 1.0 |
| Exclusive Or | xor | Yes | 1.0 |
| Exclusive Or Byte Immediate | xorbi | Yes | 1.0 |
| Exclusive Or Halfword Immediate | xorhi | Yes | 1.0 |
| Exclusive Or Word Immediate | xori | Yes | 1.0 |
| Nand | nand | Yes | 1.0 |
| Nor | nor | Yes | 1.0 |
| Equivalent | eqv | Yes | 1.0 |
| Select Bits | selb | Yes | 1.0 |
| Shuffle Bytes | shufb | Yes | 1.0 |

## A.5 SPE HINT-FOR-BRANCH INSTRUCTIONS

**Table 57: SPE Hint-for-Branch Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Hint for Branch | hbr | Yes | 1.0 |
| Hint for Branch | hbra | Yes | 1.0 |
| Hint for Branch Relative | hbrr | Yes | 1.0 |

## A.6 SPE SHIFT AND ROTATE INSTRUCTIONS

**Table 58: SPE Shift and Rotate Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Shift Left Halfword | shlh | Yes | 1.0 |
| Shift Left Halfword Immediate | shlhi | Yes | 1.0 |
| Shift Left Word | shl | Yes | 1.0 |
| Shift Left Word Immediate | shli | Yes | 1.0 |
| Shift Left Quadword by Bits | shlqbi | Yes | 1.0 |
| Shift Left Quadword by Bits Immediate | shlqbii | Yes | 1.0 |
| Shift Left Quadword by Bytes | shlqby | Yes | 1.0 |
| Shift Left Quadword by Bytes Immediate | shlquyi | Yes | 1.0 |
| Shift Left Quadword by Bytes from Bit Shift Count | shlqbybi | Yes | 1.0 |
| Rotate Halfword | roth | Yes | 1.0 |
| Rotate Halfword Immediate | rothi | Yes | 1.0 |
| Rotate Word | rot | Yes | 1.0 |
| Rotate Word Immediate | roti | Yes | 1.0 |
| Rotate Quadword by Bytes | rotqby | Yes | 1.0 |
| Rotate Quadword by Bytes Immediate | rotqbyi | Yes | 1.0 |
| Rotate Quadword by Bytes from Bit Shift Count | rotqbybi | Yes | 1.0 |
| Rotate Quadword by Bits | rotqbi | Yes | 1.0 |
| Rotate Quadword by Bits Immediate | rotqbii | Yes | 1.0 |
| Rotate and Mask Halfword | rothm | Yes | 1.0 |
| Rotate and Mask Halfword Immediate | rothmi | Yes | 1.0 |
| Rotate and Mask Word | rotm | Yes | 1.0 |
| Rotate and Mask Word Immediate | rotmi | Yes | 1.0 |
| Rotate and Mask Quadword by Bytes | rotqmby | Yes | 1.0 |
| Rotate and Mask Quadword by Bytes Immediate | rotqmbyi | Yes | 1.0 |
| Rotate and Mask Quadword Bytes from Bit Shift Count | rotqmbybi | Yes | 1.0 |
| Rotate and Mask Quadword by Bits | rotqmbi | Yes | 1.0 |
| Rotate and Mask Quadword by Bits Immediate | rotqmbii | Yes | 1.0 |
| Rotate and Mask Algebraic Halfword | rotmah | Yes | 1.0 |
| Rotate and Mask Algebraic Halfword Immediate | rotmahi | Yes | 1.0 |
| Rotate and Mask Algebraic Word | rotma | Yes | 1.0 |
| Rotate and Mask Algebraic Word Immediate | rotmai | Yes | 1.0 |

## A.7 SPE FLOATING-POINT INSTRUCTIONS

**Table 59: SPE Floating-Point Instructions**

| Name | Mnemonic | Required | Version |
|---|---|---|---|
| Floating Add | fa | Yes | 1.0 |
| Double Floating Add | dfa | Yes | 1.0 |
| Floating Subtract | fs | Yes | 1.0 |
| Double Floating Subtract | dfs | Yes | 1.0 |
| Floating Multiply | fm | Yes | 1.0 |
| Double Floating Multiply | dfm | Yes | 1.0 |
| Floating Multiply and Add | fma | Yes | 1.0 |
| Double Floating Multiply and Add | dfma | Yes | 1.0 |
| Floating Negative Multiply and Subtract | fnms | Yes | 1.0 |
| Double Floating Negative Multiply and Subtract | dfnms | Yes | 1.0 |
| Floating Multiply and Subtract | fms | Yes | 1.0 |
| Double Floating Multiply and Subtract | dfms | Yes | 1.0 |
| Double Floating Negative Multiply and Add | dfnma | Yes | 1.0 |
| Floating Reciprocal Estimate | frest | Yes | 1.0 |
| Floating Reciprocal Absolute Square Root Estimate | frsqest | Yes | 1.0 |
| Floating Interpolate | fi | Yes | 1.0 |
| Convert Signed Integer to Floating | csflt | Yes | 1.0 |
| Convert Floating to Signed Integer | cflts | Yes | 1.0 |
| Convert Unsigned Integer to Floating | cufit | Yes | 1.0 |
| Convert Floating to Unsigned Integer | cfltu | Yes | 1.0 |
| Floating Round Double to Single | frds | Yes | 1.0 |
| Floating Extend Single to Double | fesd | Yes | 1.0 |
| Double Floating Compare Equal | dfceq | No | 1.2 |
| Double Floating Compare Magnitude Equal | dfcmeq | No | 1.2 |
| Double Floating Compare Greater Than | dfcgt | No | 1.2 |
| Double Floating Compare Magnitude Greater Than | dfcmgt | No | 1.2 |
| Double Floating Test Special Value | dftsv | No | 1.2 |
| Floating Compare Equal | fceq | Yes | 1.0 |
| Floating Compare Magnitude Equal | fcmeq | Yes | 1.0 |
| Floating Compare Greater Than | fcgt | Yes | 1.0 |
| Floating Compare Magnitude Greater Than | fcmgt | Yes | 1.0 |
| Floating-Point Status and Control Register Write | fscrwr | Yes | 1.0 |
| Floating-Point Status and Control Register Read | fscrrd | Yes | 1.0 |

# Appendix B. P4080 Support Data

## B.1 E500MC PROCESSOR INSTRUCTIONS

### B.1.1 Branch and Flow Instructions

**Table 60: Branch and Flow Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Branch | b (ba bl bla) | User |
| Branch Conditional | bc (bca bcl bcla) | User |
| Branch Conditional to Link Register | bclr (bclrl) | User |
| Branch Conditional to Count Register | bcctr (bcctrl) | User |
| Integer Select | isel | User |

### B.1.2 Floating-Point Instructions

**Table 61: Floating-Point Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Floating Absolute Value | fabs[.] | User |
| Floating Move Register | fmr[.] | User |
| Floating Negative Absolute Value | fnabs[.] | User |
| Floating Negative | fneg[.] | User |

### B.1.3 Floating-Point Status and Control Register Instructions

**Table 62: Floating-Point Status and Control Register Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Move from FPSCR | mffs[.] | User |
| Move to Condition Register from FPSCR | mcrfs | User |
| Move to FPSCR Bit 0 | mtfsb0[.] | User |
| Move to FPSCR Bit 1 | mtfsb1[.] | User |
| Move to FPSCR Fields | mtfsf[.] | User |
| Move to FPSCR Fields Immediate | mtfsfi[.] | User |

### B.1.4 Floating-Point Arithmetic Instructions

**Table 63: Floating-Point Arithmetic Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Floating Add | fadd[.] | User |
| Floating Add Single | fadds[.] | User |
| Floating Divide | fdiv[.] | User |
| Floating Divide Single | fdivs[.] | User |
| Floating Multiply | fmul[.] | User |
| Floating Multiply Single | fmuls[.] | User |
| Floating Reciprocal Estimate Single | fres[.] | User |
| Floating Reciprocal Square Root Estimate | frsqrte[.] | User |
| Floating Select | fsel[.] | User |
| Floating Subtract | fsub[.] | User |
| Floating Subtract Single | fsubs[.] | User |
| Floating Multiply-Add | fmadd[.] | User |
| Floating Multiply-Add Single | fmadds[.] | User |
| Floating Multiply-Subtract | fmsub[.] | User |
| Floating Multiply-Subtract Single | fmsubs[.] | User |
| Floating Negative Multiply-Add | fnmadd[.] | User |
| Floating Negative Multiply-Add Single | fnmadds[.] | User |
| Floating Negative Multiply-Subtract | fnsub[.] | User |
| Floating Negative Multiply-Subtract Single | fnmadds[.] | User |

### B.1.5 Floating-Point Compare Instructions

**Table 64: Floating-Point Compare Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Floating Compare Ordered | fcmpo | User |
| Floating Compare Unordered | fcmpu | User |

### B.1.6 Floating-Point Rounding and Conversion Instructions

**Table 65: Floating-Point Rounding and Conversion Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Floating Convert to Integer Word | fctiw[.] | User |
| Floating Convert to Integer Word and Round to Zero | fctiwz[.] | User |
| Floating Round to Single-Precision | frsp[.] | User |

### B.1.7 Integer Instructions

**Table 66: Simple Integer Instructions**

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Add Carrying | addc[o][.] | User |
| Add Extended | adde[o][.] | User |
| Add Immediate | addi | User |
| Add Immediate Carrying | addic | User |
| Add Immediate Carrying and Record | addic. | User |
| Add Immediate Shifted | addis | User |
| Add to Minus One Extended | addme[o][.] | User |
| Add to Zero Extended | addze[o][.] | User |
| Divide Word | divw[o][.] | User |
| Divide Word Unsigned | divwu[o][.] | User |
| Multiply High Word | mulhw[.] | User |
| Multiply High Word Unsigned | mulhwu[.] | User |
| Multiple Low Immediate | mulli | User |
| Multiple Low Word | mullw[o][.] | User |
| Negate | neg[o][.] | User |
| Subtract From | subf[o][.] | User |
| Subtract from Carrying | subfc[o][.] | User |
| Subtract from Extended | subfe[o][.] | User |
| Subtract from Immediate Carrying | subfic | User |
| Subtract from Minus One Extended | subfme[o][.] | User |
| Subtract from Zero Extended | subfze[o][.] | User |
| Compare | cmp | User |
| Compare Immediate | cmpi | User |
| Compare Logical | cmpl | User |
| Compare Logical Immediate | cmpli | User |
| AND | and[.] | User |
| AND Immediate | andi. | User |
| AND Immediate Shifted | andis. | User |
| AND with Complement | andc[.] | User |
| Count Leading Zeros Word | cntlzw[.] | User |
| Equivalent | eqv[.] | User |
| Extend Sign Byte | extsb[.] | User |
| Extend Sign Half Word | extsh[.] | User |
| NAND | nand[.] | User |
| NOR | nor[.] | User |
| OR | or[.] | User |
| OR Immediate | ori | User |
| OR Immediate Shifted | oris | User |
| OR with Complement | orc[.] | User |
| XOR | xor[.] | User |
| XOR Immediate | xori | User |
| XOR Immediate Shifted | xoris | User |
| Rotate Left Word Immediate then AND with Mask | rlwinm[.] | User |

| Instruction | Mnemonic | Privilege Level |
|---|---|---|
| Rotate Left Word then AND with Mask | rlwnm[.] | User |
| Rotate Left Word Immediate then Mask Insert | rlwimi[.] | User |
| Shift Left Word | slw[.] | User |
| Shift Right Word | srw[.] | User |
| Shift Right Algebraic Word Immediate | srawi[.] | User |
| Shift Right Algebraic Word | sraw[.] | User |

## B.2 REGISTERS

### B.2.1 General Purpose Registers

**Table 67: General Purpose Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| General Purpose Registers | GPR0 - GPR31 | | User |
| Integer Exception Register | XER | 1 | User |

### B.2.2 Floating-Point Registers

**Table 68: Floating-Point Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Floating-Point Registers | FPR0 - FPR31 | | User |
| Floating-Point Status and Control Register | FPSCR | | User |

### B.2.3 Branch Registers

**Table 69: Branch Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Condition Register | CR | | User |
| Link Register | LR | 8 | User |
| Count Register | CTR | 9 | User |

### B.2.4 Branch Unit Control and Status Register

**Table 70: Branch Unit Control and Status Register**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Branch unit control and status register | BUCSR | 1013 | Hypervisor |

### B.2.5 Hardware Implementation Dependent Register

Table 71: Hardware Implementation Dependent Register

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Hardware implementation dependent register 0 | HID0 | 1008 | Hypervisor |

### B.2.6 L1 Cache Registers

Table 72: L1 Cache Registers

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| L1 cache configuration register 0 | L1CFG0 | 515 | User RO |
| L1 cache configuration register 1 | L1CFG1 | 516 | User RO |
| L1 cache control and status register 0 | L1CSR0 | 1010 | Hypervisor |
| L1 cache control and status register 1 | L1CSR1 | 1011 | Hypervisor |
| L1 cache control and status register 2 | L1CSR2 | 606 | Hypervisor |

### B.2.7 L2 Cache Registers

Table 73: L2 Cache Registers

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| L2 cache configuration register 0 | L2CFG0 | 519 | User RO |
| L2 cache control and status register 0 | L2CSR0 | 1017 | Hypervisor |
| L2 cache control and status register 1 | L2CSR1 | 1018 | Hypervisor |
| L2 cache error disable | L2ERRDIS | 725 | Hypervisor |
| L2 cache error detect | L2ERRDET | 991 | Hypervisor |
| L2 cache error interrupt enable | L2ERRINTEN | 720 | Hypervisor |
| L2 cache error control | L2ERRCTL | 724 | Hypervisor |
| L2 cache error address | L2ERRADDR | 722 | Hypervisor |
| L2 cache error extended address | L2ERREADDR | 723 | Hypervisor |
| L2 cache error capture data low | L2CAPTDATALO | 989 | Hypervisor |
| L2 cache error capture data high | L2CAPTDATAHI | 988 | Hypervisor |
| L2 cache error capture ECC syndrome | L2CAPTECC | 990 | Hypervisor |
| L2 cache error attribute | L2ERRATTR | 721 | Hypervisor |
| L2 cache error injection control | L2ERRINJCTL | 987 | Hypervisor |
| L2 cache error injection mask low | L2ERRINJLO | 986 | Hypervisor |
| L2 cache error injection mask high | L2ERRINJHI | 985 | Hypervisor |

### B.2.8 MMU Registers

**Table 74: MMU Registers**

| Register | Mnemonic | SPR # | Privilege Level |
|---|---|---|---|
| Logical Partition ID register | LPIDR | 338 | Hypervisor |
| Process ID register | PID | 48 | Guest supervisor |
| MMU control and status register 0 | MMUCSR0 | 1012 | Hypervisor |
| MMU configuration register | MMUCFG | 1015 | Hypervisor RO |
| TLB configuration register 0 | TLB0CFG | 688 | Hypervisor RO |
| TLB configuration register 1 | TLB1CFG | 689 | Hypervisor RO |
| MMU assist register 0 | MAS0 | 624 | Guest supervisor |
| MMU assist register 1 | MAS1 | 625 | Guest supervisor |
| MMU assist register 2 | MAS2 | 626 | Guest supervisor |
| MMU assist register 3 | MAS3 | 627 | Guest supervisor |
| MMU assist register 4 | MAS4 | 628 | Guest supervisor |
| MMU assist register 5 | MAS5 | 339 | Hypervisor |
| MMU assist register 6 | MAS6 | 630 | Guest supervisor |
| MMU assist register 7 | MAS7 | 944 | Guest supervisor |
| MMU assist register 8 | MAS8 | 341 | Hypervisor |
| External PID load context | EPLC | 947 | Guest supervisor |
| External PID store context | EPSC | 948 | Guest supervisor |

### B.2.9 Performance Monitoring Registers

**Table 75: Performance Monitoring Registers**

| Register | Mnemonic | PMR # | Privilege Level |
|---|---|---|---|
| *Global control register 0 | PMGC0 | 400 | Guest Supervisor |
| | UPMGC0 | 384 | User RO |
| *Performance monitor counter 0 | PMC0 | 16 | Guest Supervisor |
| | UPMC0 | 0 | User RO |
| *Performance monitor counter 1 | PMC1 | 17 | Guest Supervisor |
| | UPMC1 | 1 | User RO |
| *Performance monitor counter 2 | PMC2 | 18 | Guest Supervisor |
| | UPMC2 | 2 | User RO |
| *Performance monitor counter 3 | PMC3 | 19 | Guest Supervisor |
| | UPMC2 | 3 | User RO |
| *Performance monitor local control a0 | PMLCa0 | 144 | Guest Supervisor |
| | UPMLCa0 | 128 | User RO |
| *Performance monitor local control a1 | PMLCa1 | 145 | Guest Supervisor |
| | UPMLCa1 | 129 | User RO |
| *Performance monitor local control a2 | PMLCa2 | 146 | Guest Supervisor |
| | UPMLCa2 | 130 | User RO |
| *Performance monitor local control a3 | PMLCa3 | 147 | Guest Supervisor |
| | UPMLCa3 | 131 | User RO |
| *Performance monitor local control b0 | PMLCb0 | 272 | Guest Supervisor |
| | UPMLCb0 | 256 | User RO |
| *Performance monitor local control b1 | PMLCb1 | 273 | Guest Supervisor |
| | UPMLCb1 | 257 | User RO |
| *Performance monitor local control b2 | PMLCb2 | 275 | Guest Supervisor |
| | UPMLCb2 | 258 | User RO |
| *Performance monitor local control b3 | PMLCb3 | 276 | Guest Supervisor |
| | UPMLCb3 | 259 | User RO |

# Appendix C. List of Instructions that Cause VM Exits

**Table 76: All the instructions that cause VM Exits**

| ***Instruction that causes a VM Exit*** | ***Description and Conditions*** |
|---|---|
| CPUID, VMCALL, VMCLEAR, VMLAUNCH, VMPRTLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, AND VMXON | When they are executed in VMX non-root operation. |
| INVEPT, INVD, INVVPID | When they are executed in VMX non-root operation. |
| GETSEC | If CR4.SMXE[Bit 14] = 1. |
| XSAVE/XRSTOR/XSAVEOPT | not supported in Intel Core i7-860. |
| XGETBV | Specifying a reserved or unimplemented XCR in ECX causes a general protection exception |
| XSETBV | If CR4.OSXSAVE[Bit 18] = 1. |
| CTS | If CR0.TS is set in both the CR0 guest/host mask and the CR0 read shadow. |
| CLTS | If CR0.TS in both the CR0 guest/host mask and the CR0 read shadow does not match. |
| HLT | If the "HLT exiting" VM-execution control is 1. |
| IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD | Either "unconditional I/O exiting" or "use I/O bitmaps" VM-execution control is 1. |
| INVLPG | If the "INVLPG exiting" VM-execution control is 1. |
| LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR | If the "descriptor-table exiting" VM-execution control is 1. |
| LMSW, SMSW | If CR0.PE are set in both the CR0 guest/mask and the source operand but clear in CR0 read shadow. |
| MONITOR | If the "MONITOR exiting" VM-execution control is 1. |
| MOV from/to CR3/CR8 | If the "CR3-/CR8-store/load exiting" VM-execution control is 1. |
| MOV to CR0/CR4 | If the value to be set in CR0/CR4 guest/host mask doesn't match with the corresponding value in CR0/CR4 read shadow. |
| MOV DR | If the "MOV-DR exiting" VM-execution control is 1. |
| MWAIT | If the "MWAIT exiting" VM-execution control is 1. |
| PAUSE | Depends on "PAUSE exiting" and "PAUSE-loop exiting" |
| RDMSR, WRMSR | When "use MSR bitmaps" VM-execution control is 1, if the value of ECX is in neither of the ranges covered by the bitmaps or if the appropriate bit in the MSR bitmaps is 1. |
| RDPMC | If the "RDPMC exiting" VM-execution control is 1. |
| RDTSC | If the "RDTSC exiting" VM-execution control is 1. |
| RDTSCP | If the "RDTSC exiting" and "enable RDTSCP" VM-execution control are both 1. |
| RSM | If executed in SMM. |

| ***Instruction that causes a VM Exit*** | ***Description and Conditions*** |
|---|---|
| WBINVD | If the “WBINVD exiting" VM-execution control is 1. |
| CFLUSH, ENTER, MASKMOVQ, MASKMOVDQU, PREFETCH | If the “virtualize APIC accesses” VM-execution control is 1 and the instruction would access the APIC-access page. |
| INT3, INTO, BOUND, UD2 | The exceptions are selected in the exception bitmaps. |
| Task Switch instructions such as CALL, INT n, JMP, NMI, IRET | Task switches are not allowed in VMX non-root operation. |
| SYSCALL/SYSENTER | SYSCALL is only supported in 64-bit mode |
| MOV SS, POP SS, STI | If the “use TPR shadow” and “virtualize APIC accesses” VM-execution control are both 1. |

# Appendix D. Intel Instruction Summary

The following table is a list of the Intel instruction set that may cause vulnerabilities and is presented as an extension of Robin's work [Rob]).

**Table 77: Intel Instruction Extension Summary**

| Extension Name | Year Introduced | Number of Instructions | New features to the IA-32(e) architecture |
|---|---|---|---|
| SSE3 (Streaming SIMD Extensions 3) | 2004, with Pentium 4 processor which support Hyper-Threading Technology | 13 | • One instruction that improves x87 FPU floating-point to integer conversion.<br>• One instruction that provides a specialized 128-bit unaligned data load.<br>• Two instructions that provide packed addition/subtraction.<br>• Four instructions that provide horizontal addition/subtraction.<br>• Three instructions that enhance LOAD/MOVE/DUPLICATE performance.<br>• Two instructions that improve synchronization between multi-threaded agents. |
| XSAVE instructions | | 4 | • Two instructions to save or restore processor extended stated to or from memory.<br>• Two instructions that for read and write the state of the extended control register (XCR0) which is a 64-bit register specifies the set of processor states that the operating system enables on that processor. |
| AVX (Advanced Vector Extensions) | 2010, with the Sandy Bridge processor family. | 192 for AVX, 20 for FMA | • Support for 256-bit wide vectors and SIMD register set (YMM0-YMM7 in operating modes, YMM0-YMM15 in 64-bit mode).<br>• Instruction syntax support for generalized three-oprand syntax.<br>• 39 256-bit data processing instructions.<br>• 18 new data processing instructions that operate on 256-bit vectors.<br>• Using a new VEX-prefix in instruction encoding scheme.<br>• FMA extensions and enhanced floating compare instructions add support for IEEE-754-2008 standard. |
| VMX (Virtual Machine Extensions) | To support virtualization of processor hardware for multiple software environments | 12 | • Five instructions are to maintain the VMCS.<br>• Seven instructions are to manage the VMX. |

The following tables are a list of additional Intel instruction set that may cause vulnerabilities and is presented as an extension of Robin's work [Rob]).

**Table 78: Instructions that may Potentially Cause Vulnerabilities**

| Instruction | Extension | Class | Description | Sensitive | Privilege | Reason |
|---|---|---|---|---|---|---|
| MONITOR | SSE3 | sync | Set Up Monitor Address | Y | N | State of the Machine |
| MWAIT | SSE3 | sync | Monitor Wait | Y | N | State of the Machine |
| INVEPT | vmx | | Invalidate Translations Derived from EPT | Y | Y | memory system |
| INVVPID | vmx | | Invalidate Translations Based on VPID | Y | Y | memory system |
| VMCALL | vmx | | Call to VM Monitor | | | Protection system |
| VMCLEAR | vmx | | Clear Virtual-Machine Control Structure | Y | Y | sensitive structure |
| VMLAUNCH | vmx | | Launch Virtual Machine | Y | Y | state of the VM |
| VMPTRLD | vmx | | Load Pointer to Virtual-Machine Control Structure | Y | Y | sensitive structure |
| VMPTRST | vmx | | Store Pointer to Virtual-Machine Control Structure | Y | Y | sensitive structure |
| VMREAD | vmx | | Read Field from Virtual-Machine Control Structure | Y | Y | sensitive structure |
| VMRESUME | vmx | | Resume Virtual Machine | Y | Y | state of the VM |
| VMWRITE | vmx | | Write Field to Virtual-Machine Control Structure | Y | Y | sensitive structure |
| VMXOFF | vmx | | Leave VMX Operation | Y | Y | state of the machine |
| VMXON | vmx | | Enter VMX Operation | Y | Y | state of the machine |
| XSAVEOPT | AVX | | Save processer extended states optimized | Y | N | Sensitive registers |
| XSAVE | XSAVE | | Save processor extended states to memory | Y | N | State of the machine |
| XRSTOR | XSAVE | | Restore processor extended states form memory | Y | N | State of the machine |
| XGETBV | XSAVE | | Reads the state of an extended control register | Y | N | Sensitive registers |
| XSETBV | XSAVE | | Writes the state of an extended control register | Y | Y | Sensitive registers |
| SYSCALL | 64bit | | Fast call to privilege level 0 system procedures | Y | N | Protection system |
| SYSENTER | 64bit | | Fast call to privilege level 0 system procedures | Y | N | Protection system |

**Table 79: Additional Instructions that may Cause Vulnerabilities**

| ***Instructions*** | ***Reason*** |
|---|---|
| | |
| CALL | May reference the privilege level of a program or procedure to call. |
| INT n | Generate a call to exception handler or interrupt. |
| IRET/ IRETD | Return from an exception or interrupt handler back to a procedure or program. It examines the privilege level of procedures. |
| JMP | Call another procedure but not return. |
| LAR | Load the access rights form the segment descriptor. |
| LSL | Load the unscrambled segment limit from the segment descriptor. |
| MONITOR | Set Up Monitor Address. |
| MOV | May change the sensitive registers and segment registers. |
| MWAIT | Allow the processor to stop executing instructions and enter an implementation-dependent optimized state. |
| POP | may reference the privilege level of segment selector and segment descriptor. |
| POPF/ POPFD | Pop values from stack and stores the values into EFLAGS register. |
| PUSH | May access the privilege level of some registers. |
| PUSHF/ PUFHFD | Push the values of EFLAGS register into stack. |
| RET | Return the control to the return address on the top of the stack. |
| RSM | Return control from SMM (system management mode) to program or procedure. |
| SGDT | Store the contents of the global descriptor table register. |
| SIDT | Store the contents of the Interrupt descriptor table register. |
| SLDT | Store the contents of the local descriptor table register. |
| SMSW | Store some bits of CR0 to memory location or general purpose register. |
| STR | Store the segment selector from task register. |
| SYSCALL | Fast call to Privilege level 0 system procedures. |
| SYSENTER | Fast call to Privilege level 0 system procedures. |
| VERR | Query the current privilege level of code or data to find out if it is readable or not. |
| VERW | Query the current privilege level of code or data to find out if it is writable or not. |
| XRSTOR | Restore processor extended states from memory. |
| XSAVE | Save the processor extended states to memory. |
| XSAVEOPT | Save processer extended states optimized. |
| XGETBV | Reads the XCR (Extended Control Register). |
| | |

# Appendix E. A List of Terminology Comparison between Intel and AMD

Table 80 lists all the terminology differences between Intel VT-x and AMD-V.

**Table 80: Comparison of Common Terms between Intel VT-x and AMD-V**

| *Intel VT-x* | *AMD-V* | *Description* |
|---|---|---|
| VMX | SVM | Generic term used for extensions associated with VMs |
| root/non-root | host/guest | The cpu operation mode |
| VMCS | VMCB | VM Control data structure describing the behavior and state of the host and the guest |
| current-VMCS | RAX | Register containing physical address of current VM control data structure |
| VMPTRLD, VMLAUNCH | VMRUN RAX | Load VM control structure and start a guest |
| VMREAD, VMWRITE | VMLOAD, VMSAVE | Read/Write guest state information from/to the VMM |
| VMCALL | VMMCALL | Explicitly exit to the VMM |
| VM Exit | #VMEXIT | Event causing return to the VMM |
| Extended Page Table | Nested Page Table | Second-Level address translation |
| EPTP | nCR3 | Register contains entry address of EPT/NPT |

# Appendix F. A List of Virtualized Resource

System software virtualizes resources for each VM by constructing a specific VMCS. The VMCS data are organized into different fields. Every field of the VMCS is associated with a 32-bit encoding value, which is provided in an operand to VMREAD or VMWRITE when software wishes to read or write that field. All the virtualized resources are listed as follows in terms of different characteristics.

```
VIRTUALIZED RESOURCES{
    SEGMENTS:
            GUEST/HOST ES/CS/SS/DS/FS/GS/TR SELECTOR,
            GUEST LDTR SELECTOR,
            GUEST ES/CS/SS/DS/FS/GS/LDTR/TR/GDTR/IDTR BASE,
            HOST FS/GS/TR/GDTR/IDTR BASE,
            GUEST ES/CS/SS/DS/FS/GS/LDRR/TR/GDTR/IDTR LIMIT,
            GUEST ES/CS/SS/DS/FS/GS/LDTR/TR ACCESS RIGHTS,

VMCS:
    VIRTUAL PROCESSOR IDENTIFIER (VPID),
    EXECUTIVE VMCS POINTER,
    VMCS LINK POINTER,
    VIRTUAL APIC PAGE ADDRESS,
    APIC ACCESS ADDRESS,

    PIN-BASED VM-EXECUTION CONTROLS,
    PRIMARY PROCESSOR-BASED VM-EXECUTION CONTROLS,
    SECONDARY PROCESSOR-BASED VM-EXECUTION CONTROLS,

MEMORY:
    EPT POINTER,
    GUEST PDPTE0/PDPTE1/PDPTE2/PDPTE3,
    GUEST/HOST IA32 PAT,
    GUEST PHYSICAL ADDRESS,
    GUEST LINEAR ADDRESS,

DEBUG FACILITIES:
    GUEST IA32 DEBUG CONTROLLER,
    GUEST DR7,
    GUEST PENDING DEBUG EXCEPTIONS,

VM ENTRY/EXIT:
    VM ENTRY MSR LOAD ADDRESS,
    VM-ENTRY CONTROLS,
    VM-ENTRY MSR LOAD COUNT,
    VM-ENTRY INTERRUPTION-INFORMATION FIELD,
    VM-ENTRY INTERRUPTION ERROR CODE,

    VM EXIT MSR LOAD/STORE ADDRESS,
    VM-EXIT CONTROLS,
    VM-EXIT MSR STORE/LOAD COUNT,
    VM-EXIT REASON,
    VM-EXIT INTERRUPTION INFORMATION,
    VM-EXIT INTERRUPTION ERROR CODE,
    VM-EXIT INSTRUCTION INFORMATION,

    EXIT QUALIFICATION,

EXCEPTIONS:
    EXCEPTION BITMAP,
    PAGE-FAULT ERROR-CODE MASK,
    PAGE-FAULT ERROR-CODE MATCH,

    VM-INSTRUCTION ERROR,

    IDT VECTORING INFORMATION FIELD,
    IDT VECTORING ERROR CODE,

CONTROL REGISTERS:
    CR0/CR4 GUEST/HOST MASK,
    CR0/CR4 READ SHADOW,
    CR3 TARGET VALUE0/VALUE1/VALUE2/VALUE3,
    CR3 TARGET COUNT,
    GUEST/HOST CR0/CR3/CR4,
    TPR THRESHOLD,

OTHERS:
    GUEST RSP,
    GUEST RIP,
    GUEST RFLAGS,
    GUEST/HOST IA32 EFER,
    GUEST/HOST IA32 SYSENTER ESP,
    GUEST/HOST IA32 SYSENTER EIP,
    GUEST/HOST IA32 SYSENTER CS,
    ADDRESS OF IO BITMAP A/B,
    GUEST INTERRUPTIBILITY STATE,
    GUEST ACTIVITY STATE,
    TSC OFFSET,
}
```